Abstract

This thesis focuses on Hierarchical Machine Translation (HMT). HMT allows to model long distance phenomena and reordering at a cost of higher complexity of the decoder. With this thesis we reduce the complexity of HMT. In the first part we propose a series of alternative Cube Pruning (CP) algorithms that leverage on more aggressive pruning and less memory usage. Then we propose a linear time CP that solves exactly a relaxation of the decoding problem. All these algorithms can substitute the standard CP algorithm in any of its applications. In the second part of the thesis we present a novel Structured Prediction approach to HMT. The proposed model builds the structures incrementally, by choosing a single action at each step, and pruning all incompatible alternatives to that action. This approach allows translations to be constructed in an undirectional manner, thus not being constrained by the bottom-up ordering of CKY-like algorithms.

Reference


URN : urn:nbn:ch:unige-752802
DOI : 10.13097/archive-ouverte/unige:75280

Available at:
http://archive-ouverte.unige.ch/unige:75280

Disclaimer: layout of this document may differ from the published version.
Algorithms and frameworks for Tree-based Machine Translation and tree structures prediction

GESMUNDO, Andréa

Abstract
This thesis focuses on Hierarchical Machine Translation (HMT). HMT allows to model long distance phenomena and reordering at a cost of higher complexity of the decoder. With this thesis we reduce the complexity of HMT. In the first part we propose a series of alternative Cube Pruning (CP) algorithms that leverage on more aggressive pruning and less memory usage. Then we propose a linear time CP that solves exactly a relaxation of the decoding problem. All these algorithms can substitute the standard CP algorithm in any of its applications. In the second part of the thesis we present a novel Structured Prediction approach to HMT. The proposed model builds the structures incrementally, by choosing a single action at each step, and pruning all incompatible alternatives to that action. This approach allows translations to be constructed in an undirectional manner, thus not being constrained by the bottomup ordering of CKYlike algorithms.

Reference

Available at:
http://archive-ouverte.unige.ch/unige:75280

Disclaimer: layout of this document may differ from the published version.

[ Downloaded 04/11/2015 at 11:03:37 ]
Algorithms and Frameworks for
Tree-based Machine Translation
and Tree Structures Prediction

THÈSE
présentée à la Faculté des Sciences de l’Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par
Andrea Gesmundo
de Venise (Italie)

GENÈVE
2013
Doctorat ès sciences
Mention informatique

Thèse de Monsieur Andrea GESMUNDO

intitulée :

"Algorithms and Frameworks for Tree-based Machine Translation and Tree Structures Prediction"

La Faculté des sciences, sur le préavis de Monsieur J. ROLIM, professeur ordinaire et directeur de thèse (Département d'informatique), Monsieur J. HENDERSON, docteur (Département d'informatique), Madame P. MERLO, professeure associée (Département de linguistique, Faculté de lettres), Monsieur G. SATTA, professeur (Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Padova, Italia) et Monsieur D. CHIANG, professeur (Department of Computer Science, University of Southern California, United States of America), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 22 juillet 2014

Thèse - 4809 -

Le Doyen

N.B.- La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".
Algorithms and Frameworks for Tree-based Machine Translation and Tree Structures Prediction

Andrea Gesmundo
Algorithms and Frameworks for Tree-based Machine Translation and Tree Structures Prediction

PhD Dissertation in Computer Science

Department of Computer Science
University of Geneva

Andrea Gesmundo
Supervisor:
Dr. James Henderson
Department of Computer Science
University of Geneva

Committee members:
Prof. David Chiang
Department of Computer Science
University of Southern California

Prof. Paola Merlo
Department of Linguistics
University of Geneva

Prof. José Rolim
Department of Computer Science
University of Geneva

Prof. Giorgio Satta
Department of Information Engineering
University of Padua

The research work reported in this dissertation was partially supported by the Swiss National Science Foundation.

Copyright © 2013 by Andrea Gesmundo
Research in Natural Language Processing (NLP) aims to develop efficient and accurate computational models to automatically analyze and synthesize natural language. The work presented in this thesis focuses on Machine Translation (MT). The goal of the MT task is to automatically produce the translation of any given sentence. The source and target languages can be any natural language. Due to the huge dimensionality of the search space and variety of issues that each language pair involves, MT is considered one of the most challenging open tasks in NLP.

To solve this challenge, recent research in MT has developed Statistical Machine Learning (SML) techniques that exploit the large amount of available parallel corpora. The introduction of statistical models has led to a significant improvement of the state of the art. Supervised statistical models are able to extract the rules of translation and learn the parameter of the model from a corpus of parallel text, without any knowledge being explicitly encoded into it.

The Phrase-based approach is one of most successful statistical approaches. Phrase-based models base the translation process on short sequences of words called phrases. These models are efficient in terms or complexity. On the other hand they can produce a constrained reordering of the phrases, and cannot efficiently model long distance relations. Thus they perform poorly on language pairs such as English-Chinese, that require heavy reordering and handling of many long distance phenomena.

A solution to this problems is proposed by the Tree-based models. The gen-
eral approach that is shared by all the Tree-based models consists of building the translation in association with a tree structure. The tree structure allows long distance phenomena and reordering to be modeled, without interfering with the strengths of the Phrase-based approach. This advantage comes at a cost of higher complexity of the decoder. In this thesis we will focus on a specific Tree-based approach named Hierarchical Machine Translation (HMT) (Chiang 2007). HMT builds a tree structure on both target and source side using a synchronous translation grammar that encodes both the reordering and lexical translation.

With this thesis we address the problem of reducing the complexity of Tree-based models. The thesis is divided in two main parts: in the first part we work on the efficiency of the HMT decoder, improving on the standard Cube Pruning (CP) algorithm, while in the second part we present alternative training and decoding frameworks for Tree-based MT.

Cube Pruning is a fast method to explore the search space of a beam decoder. We propose a series of alternative CP algorithms that leverage on more aggressive pruning and less memory usage to achieve lower complexity and faster execution time. We first propose a set of faster CP algorithms that optimize the execution but keep the same overall complexity. Then we propose a linear time version of the algorithm that solves exactly a relaxation of the original problem. All these algorithms can substitute the standard Cube Pruning algorithm in any of its applications. The use of the CP algorithm is not limited to Machine Translation. It can be applied to many beam-search scenarios such as CKY-like algorithms. Thus the algorithms proposed in this part of the thesis can be applied and adapted to many other tasks.

In the second part of the thesis we present a novel Structured Prediction approach to HMT. The proposed model is deterministic in the sense that the decoder builds the structures incrementally, by choosing a single action at each step, and pruning all incompatible alternatives to that action. This greedy approach allows translation-derivations to be constructed in an undirectional manner, thus not being constrained by the bottom-up ordering imposed by CKY-like algorithms. The proposed model can learn to choose the best inference order. We introduce two alternative frameworks to train the parameters of the scoring function. The first proposed training framework focusses on the local context in which each action is selected. The second proposed training framework focusses on learning a
decoding policy that optimizes a global loss function.

Our experiments show that it is possible to achieve Tree-based MT with a lower complexity. We successfully improve the efficiency of the standard HMT decoder, and propose alternative Tree-based models with a different speed/accuracy balance.
## Contents

Summary iii
Acknowledgments xvii

1 Introduction 1
   1.1 Research Contributions ........................................... 4
   1.2 Thesis Outline ...................................................... 6

2 Cube Pruning 9
   2.1 Chapter Outline ..................................................... 9
   2.2 Related Work ........................................................ 10
   2.3 Cube Pruning Intuition .............................................. 11
   2.4 The Cube Pruning Problem .......................................... 13
       2.4.1 The Cube Pruning Problem with Monotonic Search Space 14
       2.4.2 Vectorial Representation ..................................... 16
       2.4.3 Monotonic Search Space ....................................... 16
       2.4.4 Monotonic Cube Pruning in Parsing .............................. 17
       2.4.5 Monotonic Cube Pruning as K-way Merge ..................... 18
       2.4.6 Generalizing from bidimensional search space to an arbitrary number of dimensions ......................... 21
       2.4.7 The Cube Pruning Problem with Approximately Monotonic Search Space ............................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5 Standard Cube Pruning</td>
<td>24</td>
</tr>
<tr>
<td>2.6 Faster Cube Pruning</td>
<td>27</td>
</tr>
<tr>
<td>2.6.1 Faster Cube Pruning 1</td>
<td>27</td>
</tr>
<tr>
<td>2.6.2 Faster Cube Pruning 2</td>
<td>33</td>
</tr>
<tr>
<td>2.6.3 Faster Cube Pruning 3</td>
<td>37</td>
</tr>
<tr>
<td>2.7 Linear Time Cube Pruning</td>
<td>39</td>
</tr>
<tr>
<td>2.7.1 Linear Cube Pruning Intuition</td>
<td>40</td>
</tr>
<tr>
<td>2.7.2 Naive Cube Pruning in Linear Time</td>
<td>42</td>
</tr>
<tr>
<td>2.7.3 Linear Cube Pruning algorithm</td>
<td>45</td>
</tr>
<tr>
<td>2.8 Experimental Comparison on Approximately Monotonic CP</td>
<td>54</td>
</tr>
<tr>
<td>2.8.1 Accuracy Comparison</td>
<td>55</td>
</tr>
<tr>
<td>2.8.2 Speed Comparison</td>
<td>58</td>
</tr>
<tr>
<td>2.8.3 Speed/Accuracy Balance</td>
<td>61</td>
</tr>
<tr>
<td>2.9 Discussion and Future Work</td>
<td>62</td>
</tr>
<tr>
<td>3 Deterministic Hierarchical Machine Translation</td>
<td>65</td>
</tr>
<tr>
<td>3.1 Chapter Outline</td>
<td>66</td>
</tr>
<tr>
<td>3.2 Related Work</td>
<td>68</td>
</tr>
<tr>
<td>3.3 HMT as Structured Prediction</td>
<td>69</td>
</tr>
<tr>
<td>3.4 Deterministic Machine Translation</td>
<td>70</td>
</tr>
<tr>
<td>3.4.1 Decoding Algorithm</td>
<td>72</td>
</tr>
<tr>
<td>3.4.2 DMT Decoding Example</td>
<td>75</td>
</tr>
<tr>
<td>3.4.3 Discriminative Scoring Function</td>
<td>77</td>
</tr>
<tr>
<td>3.5 Guided Structured Prediction</td>
<td>78</td>
</tr>
<tr>
<td>3.6 Discriminative Reinforcement Learning</td>
<td>82</td>
</tr>
<tr>
<td>3.6.1 Reinforcement Learning Review</td>
<td>82</td>
</tr>
<tr>
<td>3.6.2 Discriminative Reinforcement Learning Generic Framework</td>
<td>84</td>
</tr>
<tr>
<td>3.6.3 Application of Discriminative Reinforcement Learning to DMT</td>
<td>86</td>
</tr>
<tr>
<td>3.7 Comparison between GSP and DRL</td>
<td>89</td>
</tr>
<tr>
<td>3.7.1 Update techniques Comparison</td>
<td>90</td>
</tr>
<tr>
<td>3.8 Undirected Features</td>
<td>91</td>
</tr>
<tr>
<td>3.8.1 Local Features</td>
<td>91</td>
</tr>
<tr>
<td>3.8.2 Contextual Features</td>
<td>91</td>
</tr>
<tr>
<td>3.9 Experiments</td>
<td>94</td>
</tr>
</tbody>
</table>
List of Algorithms

1  Extension to multidimensional search space .................. 22
2  Standard Cube Pruning ........................................... 26
3  Faster Cube Pruning 1 ........................................... 30
4  Faster Cube Pruning 2 ........................................... 34
5  Faster Cube Pruning 3 ........................................... 38
6  Naive Cube Pruning .............................................. 44
7  Linear Cube Pruning ............................................. 47
8  DMT Decoding ..................................................... 73
9  GSP Training ....................................................... 80
10 Discriminative Reinforcement Learning ......................... 85
11 Discriminative Reinforcement Learning for DMT ............... 87
List of Figures

2.1 Space of the results obtained by summing an element of \( L_1 = \langle 1, 5, 9, 13 \rangle \) to an element of \( L_2 = \langle 1, 6, 8, 15 \rangle \). .................................. 12

2.2 Example of execution for the Cube Pruning algorithm. The input are the sorted lists: \( L_1 \equiv \langle 1, 5, 9, 11 \rangle \), \( L_2 \equiv \langle 1, 6, 8, 13 \rangle \). Shaded cells denote elements in the candidate queue, white cells denote elements added in the output list. .......................... 13

2.3 The monotonic search space shown in Figure 2.1 represented as a monotonic surface in a three-dimensional space. .......................... 15

2.4 Examples of execution for the algorithms SCP (a), FCP1 (b), FCP2 (c), and FCP3 (c). As input is given \( L \) with \( N = 2 \), and \( L_1 \equiv \langle 1, 5, 9, 11 \rangle \), \( L_2 \equiv \langle 1, 6, 8, 13 \rangle \) sorted lists. For each algorithm are depicted the first 5 iterations. Shaded cells denote elements in the candidate queue, white cells denote elements added in the \( k\)-best output list. .......................... 28

2.5 Space of the results obtained by summing an element of \( L_1 = \langle 1, 5, 9, 13 \rangle \) to an element of \( L_2 = \langle 1, 6, 8, 15 \rangle \). ......................... 40

2.6 Space of the results obtained by summing two lists with constant and equal derivatives. The derivative is \( \Delta = 3 \). The input lists are 
\( L_1 = \langle 1, 4, 7, 10 \rangle \) to an element of \( L_2 = \langle 0, 3, 6, 9 \rangle \). ......................... 41
2.7 Space of the results obtained by summing two lists with constant but different derivatives. The derivatives are $\Delta_1 = 6$ and $\Delta_2 = 3$. The input lists are $L_1 = \langle 1, 7, 13, 19 \rangle$ to an element of $L_2 = \langle 0, 3, 6, 9 \rangle$.

2.8 Sample of pattern repetition in a search space with one dimension having constant derivative.

2.9 A running example for the Linear Cube Pruning Algorithm.

2.10 Score variation in terms of probability with respect to SCP, for each algorithm presented, at different values of the beam size.

2.11 Run time variation relative to SCP for all the proposed CP algorithms, at a different beam sizes in the range $[1,10000]$.

2.12 Run time variation relative to LCP for all the CP algorithms, at a different beam sizes in the range $[1,10000]$, plotted on a log-scale.

2.13 Speed/Accuracy balance reached by the CP algorithms at different values of the beam size.

3.1 Example of binary synchronous-grammar for HMT.

3.2 Example of Deterministic Machine Translation decoding. The arrows point to the direction in which the synchronous-tree is expanded. Notice that the top-town connection at step 3 is fundamental to correctly disambiguate the translation for “go”.

3.3 Speed/Accuracy balance reached by the CP algorithms at different values of the beam size.
2.1 Evaluation and comparison of the accuracy with beam size = 30. For each of the algorithms presented the BLEU score (second column), the average internal search score (third column), and variation in terms of probability with respect to SCP (fourth column) are reported. .......................... 55

2.2 Evaluation and comparison of the accuracy with beam size = 300. For each of the algorithm presented are reported the BLEU score (second column), the average internal search score (third column), and variation in terms of probability with respect to SCP (fourth column). .......................... 56

2.3 Evaluation and comparison of the speed performance with beam size = 30. For each of the algorithms are reported: the average time for each sentence, the speed gain with respect to SCP, the average time spent on the Cube Pruning problem, the speed gain on CP with respect to SCP. .......................... 59

2.4 Evaluation and comparison of the speed performance with beam size = 300. For each of the algorithms are reported: the average time for each sentence, the speed gain with respect to SCP, the average time spent on the Cube Pruning problem, the speed gain on CP with respect to SCP. .......................... 60

3.1 Decoding speed comparison. .......................... 95
3.2 Training speed comparison. ........................................ 96
3.3 Accuracy comparison in terms of BLEU score. “Significance” is
the one-tail p-value. ...................................................... 98
I am indebted to many people for making my Ph.D. an unforgettable experience. I am deeply grateful to my advisor James Henderson, who has always been patient and has been a steady influence throughout my Ph.D. career. I thank Paola Merlo, head of the CLCL research group, for being encouraging and motivating at all times. Special thanks to all the passionate scientists with whom I had the pleasure to work with and learn from: Giorgio Satta, from the master at the University of Padua to the recent work on Cube Pruning; Keith Hall, from the Hadoop Hackathon to the internship at Google New York; Chris Dyer, for the titanic work done on CDEC. Miles Osborne and Nadi Tomeh for the intensive work done at the workshop on Hadoop and on the Hadoop Perceptron toolkit; Adam Lopez, for the interesting work done at the workshop on Minimum Bayes Risk decoding for Hierarchical Machine Translation; Tanja Samedzic, for the fabulous work on lemmatization; and all the people working on the COMTIS project (http://www.idiap.ch/project/comtis, funded by Swiss NSF grant CRSII22.127510): Andrei Popescu-Belis, Thomas Meyer, Bruno Cartoni, Cristina Grisot, Jacques Moeschler, Najeh Hajlaoui. Thanks to all the friends and colleagues from the department of Computer Science: Nikhil Garg, Gabriele Musillo, Joel Lang, Meghdad Farahmand. And thanks to all the friends and colleagues from the department Linguistics: Eric Wehrli, Christina Gulordava, Lonneke van der Plas, Effi Georgala, Alexis Kaufman, Anamaria Bentea, Lorenza Russo, Richard Zimmermann, Yves Sherrer, Sharid Loiciga. And finally, special thanks to the committee members for taking the time to review the thesis and for the influence.
tial feedbacks: Giorgio Satta, David Chiang, José Rolim, Paola Merlo and my supervisor James Henderson.
Chapter 1

Introduction

The recent application of statistical techniques to Machine Translation (MT) has led to a significant advancement of the state of the art of automatic language translation systems. Despite the translations produced by these models are still less accurate than those produced by human professional translators, their accuracy has improved to the point that some of these statistical models have been successfully proposed for practical uses, and in some cases resulting in commercial products.

Before the advent of statistical models, the state of the art was represented by models relying on manually encoded translation grammars. The research on these models was motivated by the belief that the rules of the translations for any language pairs could be fully analyzed and encoded into a set of formal rules to be executed by a computer program.

After an initial phase of enthusiasm for the success achieved in translating simple structures, the community soon realized that the MT task requires dealing with a higher level of ambiguity. The solution of the MT problem requires solving subtasks such as choosing the translation of lexical entities and word ordering. The solution of these subtasks involves an amount of irregularity and complexity that could never be fully analyzed.

Statistical models offer a natural solution to this problem. Such models are designed to discover the rules of translation automatically. Given a large corpus of text in the source language paired with its translation in the target language, a statistical model will:
• Produce a word alignment for each sentence pair,

• Extract a translation grammar,

• Learn the parameters of the model with a training process that often involves complex Machine Learning techniques.

The result is an automatic system capable of producing translations without any knowledge being explicitly encoded into it. Normally, statistical models can be generalized for any language pair, provided a parallel corpus for the specific language pair is available.

The statistical models for MT available nowadays can be classified in three main classes: Word-based models, Phrase-based models and Tree-based models. The earliest models were Word-based. The basic idea of Word-based models is to divide the translation process into a lexical translation step where each word is translated, and a reordering step where the words are reordered according to the target language.

Phrase-based models extend the intuition of the Word-based models. Phrase-based models base the translation on phrases rather than single words. A phrase is any sequence of words, not necessarily syntactically motivated nor matching with a syntactic constituent. Phrase-based models translate and reorder phrases instead of words. Basing the translation on phrases results in a stronger conditioning on the lexical context and leads to more coherent and fluent translations. On the other hand, Phrase-based models can produce a limited reordering of the phrases. Thus they perform poorly on language pairs that requires long distance reordering. Furthermore, Phrase-based models can robustly perform translations that are localized to phrases that are common enough to have been observed in training. But the model cannot perform well with infrequent phrases and long phrases due to sparsity (Koehn et al. 2003). Thus phenomena that are dependent on long distance relations and cannot be represented in a short phrase are not handled well by Phrase-based models.

Most recently, Tree-based models have been proposed. Tree-based MT models can be classified in sub-categories such as: Hierarchical models, Tree-to-string models, String-to-tree models, Tree-transducers models. The general approach that is shared by all the Tree-based models, consists in building the translation in association with a tree structure. This allows to model long distance phenom-
ena without interfering with the strengths of the Phrase-based approach. The general approach that is shared by all the Tree-based models, consists in building the translation in association with a tree structure. The tree structure can be built either on the source side or target side or both, and can be built using different kinds of grammars. The main function of the tree-structure is to encode a reordering model. The reordering produced by the tree structure is more complex than the one produced by Word-based and Phrase-based models, and it can handle long distance reordering.

The idea of building the translation with a tree structure has been proposed to more closely model the underlying recursive structure of languages, and produce more syntactically meaningful translations. On the other hand, the non-syntactically motivated Phrase-based models have linear decoding time in practice. For Tree-based models, decoding is not linear with respect to sentence length, unless the reordering range is constrained. This higher complexity of Tree-based model decoding leads to a higher execution time. Despite this efficiency issue, Tree-based models produce translations of significantly higher quality for language pairs that require complex word reordering such as English-Chinese, and it is considered the state of the art for such languages pairs. In this thesis we will focus on a specific Tree-based approach named Hierarchical Machine Translation (HMT) (Chiang 2007).

The decoding and training techniques applied to Tree-based models can be extended for generic structured prediction tasks. For example the Cube Pruning (CP) Algorithm or the Forest Rescoring Framework (Chiang 2007; Huang and Chiang 2007) have been initially proposed for Tree-based MT but have been later applied with success to other structured prediction tasks such as parsing or word alignment. And vice-versa, general techniques that have been designed for structured-prediction can be ported to Tree-based MT. Examples could be the decoding algorithm that have been ported from parsing, such as CKY decoding or more general the semiring parsing framework (Goodman 1999). We will discuss in details some of these connections that have been drawn between Tree-based MT and structured prediction. During the development of the thesis we will introduce new frameworks and algorithms, despite the fact that those have been designed explicitly for Tree-based MT, we will formalize them in general terms so that they can be ported to any structured prediction task.
Chapter 1. Introduction

The main focus of this thesis is to address the problem of reducing the complexity of Tree-based models. We tackle this issue by following two distinct lines of work:

1. Propose faster alternatives to the Cube Pruning algorithm with the aim to improve the overall efficiency of the HMT decoder,

2. Propose alternative training and decoding frameworks for Tree-based MT.

Our results show that it is possible to achieve Tree-based MT with a lower complexity. We successfully improve the efficiency of the standard HMT decoder, and propose alternative Tree-based models with a different speed/accuracy balance.

1.1 Research Contributions

This thesis contains several research contributions.

- We give a formalization of the Cube Pruning task and of the Cube Pruning algorithm that is as generic as possible to ease the application of CP in new contexts and tasks. The original definition of the Cube Pruning algorithm is tied to the Tree-based Machine Translation problem. In practice, CP can be applied to many other tasks: Phrase-based Machine Translation (Pust and Knight 2009), Parsing (Huang and Chiang 2005), Sentence Alignment (Riesa and Marcu 2010), and even non-NLP tasks such as Biological Network Alignment (Patro 2012). In general, CP can be potentially applied to all the tasks combining inexact beam decoding with dynamic programming. Thus, with the aim of ease its application within different contexts, we propose task-independent formalizations of the Cube Pruning problem and algorithm for both monotonic and approximatively-monotonic settings.

- We show how the Cube Pruning problem can be formalized as a special case of the K-way Merge problem. Relating CP with a classic algorithmic problem such as K-way merge can be helpful in finding new applications for the line of work we are presenting, and can also give new intuitions on possible extensions.

- The original CP algorithm was described for a three dimensional search space. We define a general technique that, given any algorithm solving the
1.1. Research Contributions

CP problem constrained to a fix number of dimension, allows to extend that
algorithm to solve the CP problem with a search space having any number
of dimensions.

- We propose a set of algorithms named Faster Cube Pruning (FCP) algo-
rithms. FCPs improve over the standard CP algorithm in terms of speed
and memory usage, leveraging on a more aggressive pruning of the search
space and optimizing the use and maintenance of data structures. For each
of the algorithms we provide a proof that they return the exact solution in
a monotonic search space setting. The experiments on the HMT task show
that FCPs algorithms speed up the decoding algorithm without reducing
the accuracy.

- We develop an approximate CP algorithm running in linear time named
Linear Cube Pruning (LCP). We provide a proof that LCP returns the exact
solution for a relaxed version of the original CP problem. If we consider
that CP is a special case of the K-way Merge problem, the discovery of the
LCP algorithm implies that there exists a linear time solution for a subset of
the K-way Merge problems. The experiments applying the LCP algorithm
to HMT show an actual asymptotic time advantage balanced by a bounded
loss in accuracy.

- We present a formalization of the HMT approach that explicitly addresses
the task with the general terms of structured prediction. This formalization
ease the porting of techniques defined for other structured prediction tasks
to the HMT framework and vice versa.

- We introduce a new decoding framework for Tree-based MT. This frame-
work presents a totally new approach to MT. It applies for the first time
undirectional search joint with deterministic/greedy decoding. This decod-
ing approach allows the conditioning on non-bottom-up features and lets the
decoder choose the best inference order. The experiments on the HMT task
show that this decoding approach is significantly faster than CKY-based
bottom-up decoding.

- We define a discriminative training framework to learn the parameters of
the novel deterministic MT decoder. We prove that training the model
Chapter 1. Introduction

in a discriminative fashion to match the sequence of correct actions individually without optimizing a global loss function can produce results of good quality even if a greedy search approach is used. We compare with similar techniques used for different structured prediction tasks discussing difference and similarities.

• Furthermore, we apply Reinforcement Learning (RL) techniques to define an additional training framework for the proposed deterministic MT decoder. This RL based training approach focuses more on optimizing a global loss function using Reinforcement Learning techniques. This training framework is defined in a general terms, such that it can be applied to train a discriminative scoring function for any structured prediction task.

1.2 Thesis Outline

The thesis is divided in two main parts: in the first part we describe the work done on improving the efficiency of the HMT decoder focusing on the Cube Pruning algorithm, while in the second part we present new training and decoding frameworks for Tree-based MT.

Chapter 2 contains the line of work relative to Cube Pruning. We start by introducing the topic in sections 2.2 and 2.4. We give a general formalization of the Cube Pruning problem in monotonic search space in sections 2.4.1, 2.4.2 and 2.4.3. In sections 2.4.4 and 2.4.5, we show how CP relates to well known problems such as Parsing and K-way Merge. In section 2.4.6, we describe a technique allowing to generalize a given CP algorithm for a search space with any number of dimensions. CP pruning is often applied to approximately-monotonic search spaces, thus we discuss how the algorithm is applied in these contexts as a heuristic solution in section 2.4.7. In section 2.5, we describe in detail the standard CP algorithm using a formalism consistent with the following presentation. In section 2.6, we present the set of algorithms that improve over standard Cube Pruning in terms of runtime but keeping the overall asymptotic complexity of $O(n \log n)$. Furthermore, in section 2.7, we present a linear time algorithm that solves exactly a relaxation of the original CP problem and that can be applied as heuristic solution. In section 2.8, we apply the proposed CP algorithms to the task of MT, and compare them to the standard Cube Pruning algorithm.
1.2. Thesis Outline

In Chapter 3 we present a novel Structured Prediction approach to HMT based on deterministic decoding and discriminative training. In section 3.2 and 3.3, we introduce the core concepts and formalisms on which we base the design of our models, and we discuss the related works that have inspired or introduced the novel ideas that we are introducing in Tree-based Machine Translation. In section 3.4 we introduce a novel deterministic undirectional approach for HMT. Then, we introduce two alternative frameworks to train the parameters of the scoring function. In section 3.5, we describe the first proposed training framework, that focusses optimizing each individual action selection given the local context. While in section 3.6, we describe the second proposed training framework, that focusses on learning a decoding policy that optimizes a global loss function. We discuss and compare the two proposed frameworks in section 3.7. In section 3.8 we show how the features designed for standard bottom-up HMT can be adapted to the undirected approach, and we introduce a new feature from the new class of undirected features. And finally in section 3.9, we test the proposed models and compare them with standard HMT.

In Chapter 4, we give a literature review of the publications that have been most influential for the lines of work described in this dissertation. Chapter 5 contains conclusions and future work and future research plans.
In this chapter we present a series of alternative algorithms to solve the Cube Pruning (CP) problem. The proposed algorithms leverage an aggressive pruning and little memory usage to achieve lower complexity and faster execution time. Some of the proposed algorithms have a lower asymptotic complexity than the standard CP algorithm. Any of the proposed algorithms can substitute the standard Cube Pruning algorithm in any of its applications, offering a different balance between speed and accuracy. From now on, we refer to the original CP algorithm as “Standard Cube Pruning” (SCP) to distinguish it from the algorithms we propose.

Any problem whose solution involves the application of the SCP algorithm can take advantage of the improvements described in this chapter. Either in monotonic search spaces or in search spaces with constant slope dimensions or in cases where this conditions hold approximately such as MT.

2.1 Chapter Outline

In Section 2.2 we review the history of the CP problem. From its origin to the recent adaptations to various tasks.

In Section 2.4, we give a generalized formalization of the Cube Pruning Problem for the Monotonic Search Space case, and then we extend it to the Approximately Monotonic Search Space case. Furthermore we describe how the CP
Chapter 2. Cube Pruning

problem relates to other known problems and applications such as parsing and K-way merge.

In Section 2.5, we review Standard Cube Pruning solutions based on the Lazy Algorithm, and show that it runs with a complexity of $O(k \log(k))$, where $k$ is the beam size.

In Section 2.6, we present a set of algorithms that improves over SCP. These algorithms keep $O(k \log(k))$ overall complexity. But they optimize different aspects of the selection and maintenance of candidate elements, and allow to inspect a smaller portion of the search space. We refer to these algorithms as Faster Cube Pruning (FCP) algorithms. We prove that FCP algorithms retrieve the same $k$-best as SCP in the monotonic search space case.

In Section 2.7, we present a linear time CP approximation algorithm (LCP), that has $O(k)$ complexity. LCP returns the same $k$-best as SCP in the cases where all dimensions but one have constant slope, otherwise it returns an approximation. In general, the use of LCP instead of SCP leads to an asymptotic reduction of the complexity at a cost of bounded loss in accuracy.

Finally, in Section 2.8, we test and compare empirically all the presented algorithms on an application having approximately monotonic search space: a standard Machine Translation task with Language Model (LM) features. We measure which accuracy/speed balance can be achieved with the different algorithms in different settings, show evidences of the LCP asymptotic speed advantage, and discuss in detail the empirical results.

2.2 Related Work

Part of the content of this chapter has been presented in previous publications. The first two Faster Cube Pruning algorithms appeared in (Gesmundo and Henderson 2010). While, the Linear Time Cube Pruning algorithm was presented for the first time in (Gesmundo et al. 2012).

Cube Pruning is a fast method to explore the search space of a beam decoder. Cube Pruning (CP) first appeared in Chiang (2007) as an adaptation of the Lazy Algorithm presented in Huang and Chiang (2005) to the Hiero Model (Chiang 2005). The CP algorithm quickly gained popularity in the Machine Translation (MT) community. But the use of this algorithm is not necessarily limited to Ma-
2.3. Cube Pruning Intuition

In this section we give an intuitive explanation of the Cube Pruning problem and standard solution with a simple example.

Assume we are given two ordered lists of numbers: $L_1 = \langle 1, 5, 9, 13 \rangle$ and $L_2 = \langle 1, 6, 8, 15 \rangle$. And we are asked to find the smallest number that can obtained by summing a number from the first list with a number from the second list. For example if we select 13 from the first list and 15 from the second, the result is 28. 28 is clearly not the smallest result since any other pair gives a smaller result. Indeed, finding the smallest result is trivial, since the input lists are already sorted, that implies that the smallest result is given by summing the first element of each list.

Assume now that we are asked to find sorted list containing the three smallest results. In other words, we are asked to find the smallest three numbers in the space of results of the sum of 2 numbers selected from each of the two lists, and
Chapter 2. Cube Pruning

Figure 2.1: Space of the results obtained by summing an element of $L_1 = (1, 5, 9, 13)$ to an element of $L_2 = (1, 6, 8, 15)$.

return them in a sorted list. In Figure 2.1 we picture the search space generated for this example problem. The solution is the ordered list containing the 3 best elements in the search space.

A brute-force solution would be to compute all the possible results, sort them, and then return the first three. Since the search space contains $n^2$ numbers that need to be sorted, the brute-force solution would run with complexity $O(n^2 \log n)$, where $n$ is the input lists length.

The standard Cube Pruning algorithm is able to solve the same problem in $O(k \log k)$, where $k$ is the length of the output sorted list. The algorithm starts by putting in the first place of the out list the result of the best two input element (the top left corner result in Figure 2.1). And inserting the two neighboring results in a queue of candidates (the result on the right and the result under the top left corner result). Then the algorithm iterates by moving the best result of the queue into the out list and inserting in the queue the neighbors of the moved result. In figure 2.2 we picture the state of the algorithm at the end of the first three iterations. The results with white background are already in the out list. While the results with gray background are currently in the queue. The empty cells represents results that are not required to be computed. Notice that the algorithm finds the best three elements by computing only a subset of the search space. Also notice that the out elements are already inserted in order in the list. So there is no need of a sort operation on the out list. The $\log k$ factor is instead due to the extraction of the best element of the queue at each iteration.
2.4. The Cube Pruning Problem

In this section, we give a generic formalization of the Cube Pruning Problem, first stating the problem for the monotonic search space case and then extending it to the approximately monotonic search space case. We keep the distinction between monotonic case and approximately monotonic case because:

- The monotonic case allows us to introduce and describe precisely the new CP algorithms.
- There are tasks that allow the direct application of CP with monotonicity, such as CFG parsing with no contextual features.
- When the monotonicity of the search space holds, we can prove additional properties of the CP algorithms.
- Once the algorithm is formalized for the monotonic case, its extension to the approximately monotonic case is straightforward.
- The extension to the approximately monotonic case allows the application of CP algorithms to tasks that have no strictly monotonic search space, such as MT with LM features.

Furthermore, we show how the CP problem finds application in well known NLP tasks, and how it relates to known formalisms.
2.4.1 The Cube Pruning Problem with Monotonic Search Space

In this sub-section, we describe the Cube Pruning problem in monotonic search space. Formally, we can state the problem as follows:

Problem 1. Cube Pruning Problem in Monotonic Search Space:

The input of the problem is:

• A domain set, \( X \).

• An ordering function, \( \odot \), over \( X \).

• An operator function, \( \oplus : X^n \mapsto X \). \( \oplus \) has the monotonic property, so that if \( x', x'', \hat{x} \in X \) and \( x' \odot x'' \) then \( \oplus(x', \hat{x}) \odot \oplus(x'', \hat{x}) \).

• A set of lists, \( L \equiv \{ L_n | 1 \leq n \leq |L| \} \). Each list, \( L_n \equiv \langle x_{n,1}, x_{n,2}, \cdots, x_{n,|L_n|} \rangle \), has values in \( X \): \( x_{n,i} \in X \ \forall \ 1 \leq i \leq |L_n| \); and is ordered: so that if \( x_{n,i}, x_{n,j} \in L_n \) and \( i < j \) then \( x_{n,i} \odot x_{n,j} \).

• The number of output elements, \( k \).

The solution of the problem is:

• The ordered \( k \)-best list, consisting of the first \( k \) elements of \( P \). Where \( P \) is the ordered list of all the results of the \( \oplus \)-function over \( n \) elements selected from each of the \( n \) ordered lists in \( L \), including repetitions:

\[
P \equiv \langle \oplus(x_{1,i_1}, x_{2,i_2}, \cdots, x_{|L|,i_{|L|}}) | 1 \leq i_n \leq L_n \rangle \quad (2.1)
\]

We can define formally the output \( k \)-best list as:

\[
k\text{-best} \equiv \langle x_i | 1 \leq i \leq k, x_i \in P, \forall \hat{x} \in (P - k\text{-best}), x_i \odot \hat{x} \rangle \quad (2.2)
\]

Informally, this problem applies to scenarios in which a set of solutions, \( L \), for a set of \( |L| \) input sub-problems is given, and it is required to compute the \( k \)-best solutions for the new problem representing the combination of the sub-problems.
Example 1.
Assume we are given two ordered lists of real numbers: \( L_1 = \langle 1, 5, 9, 13 \rangle \) and \( L_2 = \langle 1, 6, 8, 15 \rangle \). And we are asked to find the four best results of the sum of 2 numbers selected from each of the two lists. We can formalize the given problem as a CP problem in monotonic search space:

- \( \mathcal{X} \) is the set of real numbers, \( \mathbb{R} \).
- \( \ominus \) is the natural ordering function over \( \mathbb{R} \), \( \leq \).
- \( \oplus \) is the sum over \( \mathbb{R} \), \( + \). Notice that + has the monotonic property.
- \( \mathcal{L} = \{L_1, L_2\} \).
- \( k = 4 \).

In Figure 2.1 we picture the search space generated for this example problem. The solution is the ordered list containing the 4 best elements in the search space: \( k\text{-}best = \langle 2, 6, 7, 9 \rangle \).
2.4.2 Vectorial Representation

We introduce the vectorial representation notation. This new notation makes it easier to describe new concepts, properties and proofs.

We denote each element:

\[ \hat{x}(\hat{x} = \phi(v)) \}

with \( \hat{x} = \phi(v) \), where \( v \) is the vector of the indices of the elements in the lists \( L_n \), \( v \equiv \langle i_1, i_2, \cdots, i_{|L|} \rangle \). We can interpret \( v \) as coordinates in a \( |L| \)-dimensional space, that is our search space \( S \). With this notation, \( \phi(\cdot) \) can be interpreted as a function that maps points in the search space \( S \) to values in \( X \). We define \( \mathbf{1} \) to be the vector of length \( |L| \) whose elements are all 1, and define \( \mathbf{b}^i \) to be the vector of length \( |L| \) whose elements are all 0 except for \( b^i_i = 1 \).

In Example 1, we have that \( \phi(\mathbf{1}) = 2 \) and \( \phi(\langle 2, 1 \rangle) = 6 \). Figure 2.1 can be interpreted as the discrete bidimensional search space, plotting the \( \phi(\cdot) \) value for each element in the space.

2.4.3 Monotonic Search Space

We can use the vectorial representation notation to formally define the concept of monotonicity of the search space:

**Definition 1.** A search space \( S \) is monotonic iff

\[ \phi(v) \leq \phi(u) \quad \forall \ v, u \in S \ \text{ s.t. } v_i \leq u_i, \ 1 \leq i \leq |L| \]

In words, a search space is monotonic if any element represented by a vector \( v \) has a value that is not “worse” than the values of all the elements that are represented by vectors having all the coordinates greater or equal to the corresponding coordinate of \( v \).

In Example 1, consider \( v = \langle 2, 2 \rangle \), its value \( \phi(v) = 11 \) is not “worse” than the values of all the elements in the square having top left vertex in \( v \) and bottom right vertex in \( \langle 4, 4 \rangle \). The bidimensional search space of Example 1 is represented in Figure 2.3 as a monotonic surface in a three-dimensional space,
where the x-y-plane maps to the search space, while the z-axis represents the $\phi(\cdot)$ values dimension.

The monotonicity of the search space is implied by having a monotonic operator, $\oplus$, and ordered generating lists, $\mathcal{L}$.

### 2.4.4 Monotonic Cube Pruning in Parsing

In this section we show the correspondence between Cube Pruning Problem in Monotonic Search Space and the task of Parsing. Consider a Probabilistic Context Free Grammar (PCFG) parser that builds tree structures using the standard CKY algorithm. The PCFG parser is given as input:

- A Context Free Grammar, $G$, that consists in a set of rules, $r$, to each rule is associated a cost $\text{Cost}(r)$.
- A sentence, $W = \langle w_1, w_2, ..., w_{|W|} \rangle$.

The parser is required to output the $k$-best tree structures, $t$, with lowest cost among all the tree structures that generate the input sentence $W$. Where, the cost of a tree structure is defined as the sum of the cost of all the rules composing the tree structure:

$$\text{Cost}(t) = \sum_{r \in t} \text{Cost}(r) \quad (2.5)$$

We assume that the parser builds the tree structures using the CKY algorithm. CKY is a dynamic programming algorithm that builds the tree structures in a bottom-up fashion. At each intermediate step, the CKY algorithm finds the $k$-best sub-trees generating the span $[i,j]$ of the input sentence $W$, $1 \leq i < j \leq |W|$. When the span $[i,j]$ is processed, the $k$-best sub-trees generating any of the sub-spans of $[i,j]$ are already known, since the algorithm proceeds in a bottom-up fashion. To show the correspondence with the CP problem, assume we are given a rule instantiation: $r : A \rightarrow B C$, where $A$ covers $[i,j]$, $B$ covers the sub-span $[i,h]$ and $C$ covers $[h+1,j]$. The list of the $k$-best subtrees is already known for each of the 2 sub-spans. Then, the problem of finding the $k$-best trees covering $[i,j]$, constructed by connecting $r$ to sub-trees covering sub-spans $[i,h]$

---

1If the rules are associated with a probability, then the cost is defined as the negative logarithm of the probability.
and \([h + 1, j]\), can be formalized as the Cube Pruning Problem in Monotonic Search Space making the following associations:

- \(\mathcal{X}\) is the domain of the cost values, for example \(\mathbb{R}\).
- \(\otimes\) is the natural ordering function over \(\mathbb{R}\), \(\leq\).
- \(\oplus\) is the function computing the cost of the new tree built connecting \(r\) with the chosen sub-trees: \(\oplus(\text{Cost}(t_A), \text{Cost}(t_B)) = \text{Cost}(r) + \text{Cost}(t_A) + \text{Cost}(t_B)\)
- \(\mathcal{L}\) is the set of the lists of \(k\)-best elements of each sub-span: \(\{L_A, L_B\}\).
- \(k\) is the beam size.

Notice that the correspondence can be easily extended to non-binary rules introducing a list \(L_X\) for each additional child. Given the correspondence among the CP problem and a subproblem of PCFG Parsing, we can deduce that any algorithm solving the CP problem can be applied to solve the stated parsing subproblem. Following similar reasoning, we can find correspondences between CP and any other problem whose solution involves the use of CKY-like dynamic programming techniques.

### 2.4.5 Monotonic Cube Pruning as K-way Merge

In this section we show that the monotonic Cube Pruning problem can be formalized as a special case of K-way Merge problem. Formalizing this connection between CP and a classic Algorithmic problem such as K-way merge can be helpful in finding new applications for the line of work we are presenting, and can also give new intuitions on possible extensions.

Let us formally state the general K-way Merge problem:

**Problem 2. K-way Merge Problem:**

The input of the problem is:

- A domain set, \(\mathcal{X}\).
- An ordering function, \(\otimes\), over \(\mathcal{X}\).
2.4. The Cube Pruning Problem

- A set of lists, $\mathcal{L}^{\text{mrg}} \equiv \{L_n^{\text{mrg}} \mid 1 \leq n \leq K\}$, where $K = |\mathcal{L}^{\text{mrg}}|$. Each list, $L_n^{\text{mrg}} \equiv \langle x_{n,1}, x_{n,2}, \ldots, x_{n,|L_n^{\text{mrg}}|} \rangle$, has values in $\mathcal{X}$: $x_{n,i}^{\text{mrg}} \in \mathcal{X} \; \forall \; 1 \leq i \leq |L_n^{\text{mrg}}|$; and is ordered: so that if $x_{n,i}^{\text{mrg}}, x_{n,j}^{\text{mrg}} \in L_n^{\text{mrg}}$ and $i < j$ then $x_{n,i}^{\text{mrg}} \leq x_{n,j}^{\text{mrg}}$.

The solution of the problem is:

- The ordered list, $\mathcal{P}$, containing all the elements of all the lists in $\mathcal{L}^{\text{mrg}}$, including repetitions:

  \[ \mathcal{P} \equiv \langle x_{n,i}^{\text{mrg}} \mid 1 \leq n \leq K, 1 \leq i \leq |L_n^{\text{mrg}}| \rangle \quad (2.6) \]

  Informally, we are given $K$ sorted lists and we are required to merge them in a single sorted list. We write:

  \[ \mathcal{P} = \text{merge}_n^{K} L_n^{\text{mrg}} \quad (2.7) \]

to denote the “merge” of all the lists $L_n^{\text{mrg}}$ into a single sorted list, $\mathcal{P}$.

Now we show how to formalize the CP problem in a monotonic bidimensional search space as a K-way merge problem:

**Problem 3. Cube Pruning problem in a monotonic bidimensional search space as K-way Merge problem:**

The input of the K-way Merge problem is:

- The domain set, $\mathcal{X}$, is the same defined for the CP problem.

- The ordering function, $\odot$, over $\mathcal{X}$, is the same defined for the CP problem.

- The set of lists, $\mathcal{L}^{\text{mrg}} \equiv \{L_n^{\text{mrg}} \mid 1 \leq n \leq K\}$, is defined using the operator, $\oplus$, and the two lists, $L_1$ and $L_2$, given as input to the bidimensional monotonic CP problem. To ease the formalization, we introduce the function $\text{shift}(\mathcal{L}, \Delta) = \mathcal{L} \oplus \langle \Delta \rangle$. In words, $\text{shift}(\mathcal{L}, \Delta)$ is the sorted list whose elements are obtained by adding $\Delta$ to each element of $\mathcal{L}$, preserving the order.

Each list $L_n^{\text{mrg}}$ can be defined as:

\[ L_n^{\text{mrg}} \equiv \text{shift}(L_1, x_{2,n}), \; x_{2,n} \in L_2 \quad (2.8) \]
meaning that $L^\text{mrg}_n$ is $L_1$ shifted by the $n$-th element of $L_2$.

The solution of the problem is:

- The ordered list, $\mathcal{P}$, can be expressed as:

$$\mathcal{P} = \text{merge}^{[L_2]}_{n=1} \text{shift}(L_1, x_{2,n})$$  \hspace{1cm} (2.9)

The $\mathcal{P}$ defined for the CP problem matches the one defined for the K-way Merge. The CP problem requires to output the truncated $\mathcal{P}$ to the size of $k$, that is the $k$-best list. While K-way Merge outputs the entire $\mathcal{P}$.

Notice that $L_1$ and $L_2$ can be swapped without affecting the final output.

By mapping a monotonic CP problem, we obtain a K-way Merge problem with specific additional constraints. In the general K-way Merge, the lists $L^\text{mrg}_n$ are not required to have the same length. While for a mapped CP problem, all the $L^\text{mrg}_n$ have the same length, since those are all shifts of $L_1$. Furthermore, in the general K-way Merge, the relation between the elements in the lists is arbitrary. While for a mapped CP problem, given any two lists $L^\text{mrg}_y = \text{shift}(L_1, x_{2,y})$ and $L^\text{mrg}_j = \text{shift}(L_1, x_{2,j})$, and any position in the list $1 \leq i \leq |L_1|$, we can derive from the definitions that the following property holds:

$$x^\text{mrg}_{y,i} \oplus x_{2,j} = x^\text{mrg}_{j,i} \oplus x_{2,y}$$  \hspace{1cm} (2.10)

Where $x^\text{mrg}_{y,i}$ is the $i$-th element of the list $L^\text{mrg}_y$. This can be proved observing that from the definition of the $\text{shift}(\cdot)$ function we have that: $x^\text{mrg}_{y,i} = x_{1,i} \oplus x_{2,y}$ and $x^\text{mrg}_{j,i} = x_{1,i} \oplus x_{2,j}$.

Thus the set of all CP problems actually maps to a subclass of K-way Merge problems. This shows that the CP problem is a particular instance of the K-way Merge problem, in which all input lists are related by $K$ independent shifts. We will refer to this subclass of the K-way Merge problem as K-way Merge with shifted lists.

Having shown that CP can be formalized as a K-way Merge problem we can deduce that any algorithm solving K-way Merge can be applied to solve the CP problem. So we can use for CP any efficient algorithm used to solve the K-way Merge. The computation of the solution of the K-way merge problem takes time
2.4. The Cube Pruning Problem

\( \mathcal{O}(|P| \log(K)) \). In a standard CP application to a bottom-up beam decoder, each input list has length \( k \), and \( K = k \), then the complexity becomes \( \mathcal{O}(k^2 \log(k)) \), and by restricting the computation to the first \( k \) elements, as required by the CP problem, we can further reduce to \( \mathcal{O}(k \log(k)) \). This is the already known upper bound on the CP problem. In Section 2.7, we will address the issue of achieving an asymptotically faster algorithm by exploiting the additional constraints on the input lists.

We formalized this mapping for the bidimensional case. An elegant way to generalize to an arbitrary number of dimensions is described in Section 2.4.6.

2.4.6 Generalizing from bidimensional search space to an arbitrary number of dimensions

In this section we describe a simple technique to generalize any CP formalization/algorithm from a bidimensional search space, to a search space with an arbitrary number of dimensions. For example, in section 2.4.5 we defined the mapping from the binary CP problem to the K-way merge problem. That result can be extended for mapping from a CP problem with an arbitrary number of dimensions to a K-way merge problem. This extension can be done by applying the technique we describe in this section. Furthermore, in the following sections we will describe many algorithms that solve the CP problem and some of its relaxations. Some of these algorithms are constrained to a bidimensional search space. The extension of all these algorithms can be done by applying this generic technique. This approach renders the explanation modular, and thereby easier to read and understand, rather than having to detail a more complex version for each of these algorithms.

The extension technique is based on recursion. To give an intuition about the extension technique, assume we are given a CP problem with \( |\mathcal{L}| \) input lists. We can recursively merge \( |\mathcal{L}| - 1 \) lists into a single sorted list, and thus reducing the problem to the bidimensional case.

Formally, consider a generic CP algorithm that is defined for a bidimensional search space: \textbf{Bidimensional-GenericCP}(\( \mathcal{L} \)) : \( k \)-best, with \( |\mathcal{L}| = 2 \). We define a technique that extends the \textbf{Bidimensional-GenericCP} algorithm into a CP algorithm that can be applied to a search space with any number of dimensions: \textbf{GenericCP}(\( \mathcal{L} \)) : \( k \)-best, with \( |\mathcal{L}| \geq 2 \).
Algorithm 1  Extension to multidimensional search space

1: function GenericCP (L) : k-best
2: if |L| == 2 then
3:     return Bidimensional-GenericCP(L);
4: else
5:     $L'_2 \leftarrow$ GenericCP($\{L_2, \ldots, L_{|L|}\}$);
6:     return Bidimensional-GenericCP($\{L_1, L'_2\}$);
7: end if

Algorithm 1 defines the extension technique. lines 2-3 handle the base case by calling the bidimensional version of the algorithm, if the input set contains only 2 elements. line 5 applies the recursion by calling the extended algorithm on the $L - L_1$ set, and store the intermediate $k$-best list in $L'_2$. line 6 uses the bidimensional algorithm to compute $k$-best in the binary search space given by $L_1$ and $L'_2$. Notice that in the monotonic case, the ordering of the lists does not affect the final output. If the bidimensional algorithm is exact also the extension is exact.

If the complexity of bidimensional algorithm is $O(X)$, then the complexity of the extended algorithm is $O(|L|X)$. In many applications $|L|$ is considered a constant. Anyway, to design a CP algorithm that can be applied to an arbitrary number of dimensions without accounting for $|L|$ factor in the asymptotic complexity is a non-trivial problem, and is not possible to the best of our knowledge. Thus we can assume that applying the extension technique to an optimal algorithm does not generate a suboptimal extension. For example, if we constrain the Standard Cube Pruning algorithm to a bidimensional search space and then we apply the extension technique to this constrained algorithm, the resulting extended algorithm will have the same overall complexity as the original Standard Cube Pruning algorithm.

2.4.7 The Cube Pruning Problem with Approximately Monotonic Search Space

In this section, we describe the Cube Pruning problem in approximately monotonic search space by extending the monotonic case. In section 2.4.3 we defined the monotonic search space. In words, we can define the approximately monotonic search space as a search space that is mostly monotonic except for small
2.4. The Cube Pruning Problem

areas of non-monotonicity.

In the approximately monotonic case, the set of ordered lists $\mathcal{L}$, the ordering function $\odot$, and the monotonic operator $\oplus$ are defined as in Section 2.4.1. The main difference is that instead of defining the $\oplus$-product of the single input items: 

$$\hat{x} = x_{1,i_1} \oplus x_{2,i_2} \oplus \cdots \oplus x_{N,i_N},$$

in the approximately monotonic case we have to add a perturbation element:

$$\hat{x} = x_{1,i_1} \oplus \cdots \oplus x_{N,i_N} \oplus \Delta(i_1, \cdots, i_N) \quad (2.11)$$

Where $\Delta(i_1, \cdots, i_N)$ is the perturbation function, $\Delta(\cdot) : \mathcal{S} \rightarrow \mathcal{X}$. For brevity we define $\psi(v) = \phi(v) \oplus \Delta(v)$. Notice that $\phi(v)$ is a monotonic factor. The perturbation function $\Delta(\cdot)$ must have two properties:

- $\Delta(\cdot)$ is not monotonic.

Therefore, there can be cases in which $\phi(v) \leq \phi(u)$, but $\psi(v) \nleq \psi(u)$. In those cases the search space described by $\psi(\cdot)$ has some perturbations and it could have areas of non-monotonicity.

- The magnitude of $\Delta(v)$ must be smaller or proportional to $\phi(v)$. In cases where $|\Delta(v)| \gg |\phi(v)|$ the search space could become fully non-monotonic.

Examples of adding a perturbation function $\Delta(\cdot)$ are adding the Language Model in a Machine Translation system, or adding global features in a CKY syntactic parser. In general, the non-monotonic factor $\Delta(\cdot)$ is used to account for non-local contextual features in the scoring function.

The stated definition of the approximately monotonic search space does not give any hard constraints on the function $\psi(\cdot)$. To guarantee finding the exact $k$-best list would require exploring the entire search space. This is because the function $\Delta(\cdot)$ could be any generic function for which the two stated properties hold, and there are no constraints on the kind of distortion introduced in the search space. In practical applications, this issue is addressed by applying the Standard Cube Pruning algorithm as a heuristic algorithm to find an approximate solution in the approximately monotonic case. SCP applied to the approximately monotonic search space is still efficient and inspects only a small corner of the search space, but it returns a $k$-best list that might depart from the exact solution in two respects: it might not be a descending list, because of local variations in
the ordering of the elements; and it might not be a permutation of the exact solution, because of local variations at the end of the list.

Chiang (2007) applies the Standard Cube Pruning algorithm using a margin parameter $\epsilon$ to control the level of uncertainty about whether the true $k$-best has been found. Cube Pruning continues to search the space until the $k$-best contains $k$ elements and all the elements $v \in S$ such that $\psi(v) \circ \psi(u^*) \oplus \epsilon$ have been computed, where $u^*$ is the “worst” element in $k$-best. Thus the algorithm does not stop as soon as the list is full, but continues to search in the neighborhood of the top corner of the search space for a while. Applications with a small ratio $R = |\Delta(v)|/|\phi(v)|$ are closer to the monotonic case, and we can use a small $\epsilon$ and still have good quality results. When $R$ is big, we need to tune $\epsilon$ to a higher value, which will slow down execution because more of the space is searched. In any case, Standard Cube Pruning applied to the approximately monotonic search space is not exact and returns an approximate $k$-best list.

We can apply the same reasoning with any other algorithm that solves exactly the CP problem in the monotonic case, and apply such algorithm to the approximately monotonic case. Thus, in the following sections we are going to propose efficient algorithms that have been proved to solve exactly the monotonic CP problem, and test them empirically on the approximately monotonic case. As for SCP, the proposed algorithms output an approximate $k$-best list in the approximately monotonic case, and can use a margin parameter $\epsilon$ to control the uncertainty.

### 2.5 Standard Cube Pruning

In this section, we describe how to solve the monotonic CP problem using the Standard CP algorithm. To simplify the explanation, we define the algorithm using the Vectorial Representation formalism introduced in 2.4.2.

A brute force solution to the monotonic CP problem is to compute the $\phi(\cdot)$ values for all the elements in the search space, sort them, and pick the $k$-best. This brute force algorithm has complexity $O(n^2)$. The SCP algorithm improves on that by leveraging on the constraint of monotonicity. SCP returns the exact $k$-best without searching the entire search space. SCP has complexity $O(n \log n)$.

The SCP algorithm proceeds recursively. At each iteration a new element of
2.5. Standard Cube Pruning

the $k$-best list is found. At the first iteration, the element $\phi(1)$ is added to the $k$-best list. Because of the monotonicity constraint, it can be inferred that $\phi(1)$ is the best element in $S$. For all the iterations following the first, the next element, $v$, to be inserted in the $k$-best list is selected from a Queue of candidates, $Q$. Such $v$ is the element in $Q$ having the best $\phi(\cdot)$ value according to the ordering function $\gtrdot$. After adding the new element, $v$, to the $k$-best list, all $v$'s following elements are visited. Formally the set of following elements $F_v$ of $v$ is defined as:

$$F_v \equiv \{ u = v + b^i | 1 \leq i \leq |L|, u \in S \}$$

(2.12)

Where, $b^i$ is the vector of length $|L|$ whose elements are all 0 except for $b^i_i = 1$. An element $u$ is added into $Q$ only if it has not already been inserted in a previous iteration. This check is needed since a generic element of the search space can be in more than one set of following elements:

$$\exists u, v, v' \in S | u \in F_v \land u \in F_{v'}$$

(2.13)

The algorithm terminates when either the queue is empty or the $k$-best list has $k$ elements. So the number of iterations is $\min(|S|, k)$. Figure 2.4 (a) pictures an example for the Standard Cube Pruning for a bidimensional case $|L| = 2$.

SCP pseudocode is reported in Algorithm 2. At line 2 the queue $Q$ is initialized inserting the origin element 1. At line 3 the $k$-best list is initialized as an empty list. The main loop starts at line 4, it will terminate either when the $k$-best list is full, or when the queue is empty; this second case occurs when the search space contains less than $k$ elements. At line 5, as first step of the loop, the best element in $Q$ is popped and named $v$. At line 6 $v$ is inserted in the $k$-best. From line 7 to line 11 the algorithm iterates over all the coordinates of $v$, and adds them to $Q$. The test at line 8 checks that the value to be inserted into $Q$ is not already in $Q$. The pseudocode for the function that retrieves the neighboring elements is listed between line 13 and line 20. This function implements the definition of $F_v$. At line 14 $F_v$ is initialized as an empty set. The loop between line 15 and line 20 iterates over all the coordinates of $v$. Each following element is created at line 16 incrementing by 1 a coordinate of $v$. The new element is added in $F_v$ at line 18. The test at line 17 is needed to check that the incremented coordinate is still inside the valid range of values, and points to
Algorithm 2 Standard Cube Pruning

1: function MainLoop ($X$, $\ominus$, $\oplus$, $L$, $k$) : $k$-best
2: $Q \leftarrow \{1\}$;
3: $k$-best $\leftarrow$ empty-list;
4: while $|k$-best$| < k$ and $|Q| > 0$ do
5:   $v \leftarrow$ get-best($Q$, $\ominus$);
6:   insert($k$-best, $\phi(v)$);
7:   for all $v' \in$ FollowersOf($v$) do
8:     if $v' \notin Q$ then
9:       insert($Q$, $v'$);
10:   end if
11: end for
12: end while

13: function FollowersOf($v$): $\mathcal{F}_v$
14: $\mathcal{F}_v \leftarrow$ empty-set;
15: for $1 \leq i \leq |v|$ do
16:   $v^* \leftarrow v + b^i$
17:   if $v^*_i \leq |L_i|$ then
18:      insert($\mathcal{F}_v$, $v^*$);
19:   end if
20: end for
an actual element in $L_i$.

To find the $k$-best elements, the SCP algorithm avoids computing the $\phi(\cdot)$ value for all the elements in the search space $S$. It just needs to compute the $\phi(\cdot)$ value for those $k$-best elements, along with few other elements that remain in the queue of candidates when the algorithm terminates. Note that, for monotonic search space applications, there is no approximation: the SCP algorithm finds the exact $k$-best list, and returns it already ordered. This is not true for the approximately monotonic case as discussed in Section 2.4.7.

To improve the SCP algorithm we cannot avoid computing the $\phi(\cdot)$ values for the $k$-best elements. What can be done is to reduce the number of elements that are in the queue during execution time, and the number of leftover elements that remain in the queue at the end of the execution. A shorter queue needs less memory at run time and consumes less time when adding the new elements in order. Furthermore, minimizing the number of elements left in the queue at the end of the execution reduces further the execution time by avoiding the computation of elements that are not relevant for the final output.

## 2.6 Faster Cube Pruning

In this section we present a set of algorithms that improve over SCP in terms of speed and memory usage, leveraging a more aggressive pruning of the search space and optimizing the use and maintenance of data structures. For each of the algorithm we provide a proof that they return the exact $k$-best in a monotonic search space. We refer to this set of algorithms as Faster Cube Pruning (FCP).

### 2.6.1 Faster Cube Pruning 1

In this subsection we describe Faster Cube Pruning 1 (FCP1), the first algorithm that we propose as improvement over SCP. To introduce FCP1, let us focus on a detail of the SCP algorithm: before inserting a following element $v'$ in $Q$, SCP needs to check that $v'$ has not already been added (see Algorithm 2 line 8). At the implementation level, this check requires either a search in $Q$, that costs $O(\log |Q|)$, or an additional hash table that allows testing the presence of $v'$ in $Q$ in constant time but needs to be maintained as $Q$ changes. This test is needed because an element can be in the set of following elements of more than one
Figure 2.4: Examples of execution for the algorithms SCP (a), FCP1 (b), FCP2 (c), and FCP3 (c). As input is given \( \mathcal{L} \) with \( N = 2 \), and \( L_1 \equiv \langle 1, 5, 9, 11 \rangle \), \( L_2 \equiv \langle 1, 6, 8, 13 \rangle \) sorted lists. For each algorithm are depicted the first 5 iterations. Shaded cells denote elements in the candidate queue, white cells denote elements added in the \( k \)-best output list.

**parent element**, thus it could be inserted in \( Q \) as follower of different parent elements. Formally, we define the set of parent elements of \( \mathbf{v} \) with

\[
\mathcal{A}_\mathbf{v} \equiv \{ \mathbf{u} | \mathbf{v} \in \mathcal{F}_\mathbf{u}, \mathbf{u} \in \mathcal{S} \} \quad (2.14)
\]

We define the new FCP1 algorithm by giving an alternative definition of following elements \( \mathcal{F}_\mathbf{v} \). We define a new set of following elements \( \mathcal{F}'_\mathbf{v} \) such that \( \mathcal{A}'_\mathbf{v} \equiv \{ \mathbf{u} | \mathbf{v} \in \mathcal{F}'_\mathbf{u}, \mathbf{u} \in \mathcal{S} \} \) contains a single element for all \( \mathbf{v} \in \{ \mathcal{S} - 1 \}^2 \). We propose the following definition for \( \mathcal{F}'_\mathbf{v} \):

\[
\mathcal{F}'_\mathbf{v} \equiv \{ \mathbf{u} = \mathbf{v} + \mathbf{b}^i | \forall j < i, v_j = 1, 1 \leq i \leq |\mathcal{L}|, \mathbf{u} \in \mathcal{S} \} \quad (2.15)
\]

Where \( v_j \) is the \( j \)-th coordinate of the vector \( \mathbf{v} = \langle v_1, v_2, \cdots, v_N \rangle \). Notice that \( \mathcal{F}'_\mathbf{v} \) contains a subset of the elements in \( \mathcal{F}_\mathbf{v} \), \( \mathcal{F}'_\mathbf{v} \subseteq \mathcal{F}_\mathbf{v} \). Specifically, \( \mathcal{F}'_\mathbf{v} \) contains all and only the elements of \( \mathcal{F}_\mathbf{v} \) that have all unit values for the coordinates.

\(^2\)The set of parents of \( \mathbf{1} \) is always empty by definition.
2.6. Faster Cube Pruning

preceding the one updated. For example: \( v + b^1 \) is always added in \( F'_v \); \( v + b^2 \) is added only if \( v_1 = 1 \); and in general \( v + b^i \) is added only if \( \{v_1, v_2, \cdots, v_{i-1}\} \) are all equal to 1. In this setting, it becomes necessary to have an order of the coordinates of \( v \). This means that we need to set an ordering of the elements in \( \mathcal{L} \), and \( \mathcal{L} \) becomes an ordered list rather than a set. This can be seen as deciding an order for the dimensions of the search space \( \mathcal{S} \).

Having defined \( F'_v \), we can consider the set of parent elements of \( v \) : \( \mathcal{A}'_v \equiv \{ u | v \in F'_u, u \in \mathcal{S} \} \). Now we prove that \( \mathcal{A}'_v \) contains only one element for any \( v \neq 1 \).

**Theorem 1.** For any element \( v \neq 1 \) in the search space \( \mathcal{S} \), we have that \( |\mathcal{A}'_v| = 1 \).

**Proof.** By contradiction:
Assume that there is an element \( v \) for which \( |\mathcal{A}'_v| > 1 \).
Let \( u, w \in \mathcal{A}'_v, u \neq w \).
From the definition of \( \mathcal{A}'_v \), we have that: \( v \in F'_u \) and \( v \in F'_w \).
From the definition of \( F'_v \), we have that: \( v = u + b^i, v = w + b^j \) and we have also that \( u_x = 1 \ \forall \ x < i, w_y = 1 \ \forall \ y < j \), with \( 1 \leq i, j \leq |\mathcal{L}| \).
Then we can infer:

\[
  v = u + b^i \land v = w + b^j \Rightarrow u = w + b^j - b^i
\]  
(2.16)

To conclude we distinguish three cases:

- if \( i = j \) then:
  \( u = w + b^j - b^i = w \) and this cannot be because it is in contradiction with \( u \neq w \).

- if \( i < j \) then:
  From definition of \( b^i \) we have that: \( b^j_i = 0 \), \( b^i_i = 1 \)
  From \( i < j \) and the definition of \( F'_v \) we have that: \( w_i = 1 \)
  Therefore : \( u_i = w_i + b^j_i - b^i_i = 0 \)
  But this assignment of values cannot be because no vector \( v \in \mathcal{S} \) can have coordinate with value 0 since the indexing of the lists in \( \mathcal{L} \) start from 1 by definition.
• Similarly if $i > j$.

We showed by contradiction that $A'_v$ cannot contain two different elements. Therefore $|A'_v| \leq 1$.

Now consider any $v \neq 1$, and let $i$ be the index of the first coordinate in the sequence so that $v_i > 1$. We compute $u = v - b^i$, then from the definition of $F'$ we have that $v \in F'_u$. And we can state that for $v \neq 1$ we have that $|A'_v| > 0$.

Combining $|A'_v| \leq 1$ and $|A'_v| > 0$, it follows that $|A'_v| = 1$.

Having defined how FCP1 retrieves the following elements, it's straightforward to write the new algorithm. The pseudocode is listed in Algorithm 3. FCP1 does

**Algorithm 3** Faster Cube Pruning 1

1: function MainLoop ($X$, ⊗, ⊕, $L$, $k$) : $k$-best
2: $Q \leftarrow \{1\}$;
3: $k$-best ← empty-list;
4: while $|k$-best$| < k$ and $|Q| > 0$ do
5:   $v \leftarrow$ get-best($Q$, ⊗);
6:   insert($k$-best, $\phi(v)$);
7:  for all $v' \in$ FollowersOf($v$) do
8:     insert($Q$, $v'$);
9:  end for
10: end while
11: function FollowersOf($v$): $F'_v$
12: $F'_v \leftarrow$ empty-set;
13: for $1 \leq i \leq |v|$ do
14:   $v^* \leftarrow v + b^i$
15:  if $v^*_i \leq |L_i|$ then
16:     insert($F'_v$, $v^*$);
17:  end if
18:  if $v_i \neq 1$ then
19:     break;
20: end if
21: end for

not need to check if the neighbors are already in $Q$. Therefore the test that was done at line 8 of Algorithm 2 (SCP) is removed. This allows us to avoid the use of a hash table or the execution of a $O(log|Q|)$ search on $Q$. The new FollowersOf($v$) function is similar to the first version. Only the test at line 18 is
added to break the loop when the first coordinate in the sequence that has a value different from 1 is found. The loop break allows FCP1 to visit fewer elements than SCP, and to return a smaller set of following elements and therefore the size of $Q$ will be smaller at execution time, and furthermore less time is spent in the loop. Since most of the elements in the search space have first coordinate with value different from 1, the loop at line 13 will execute only the first round in most of the cases. For example, in a bidimensional search space of size $n^2$, line 13 does not break at the first round only for $n$ elements in the search space. In general, in a $N$ dimensional search space, line 13 does not break at the first round for $n^{N-1}$ elements in the search space, while it does break at the first round for the remaining $n^N - n^{N-1}$ elements. Figure 2.4 (b) illustrates an example for Algorithm 3 (FCP1) for a bidimensional case $N = 2$, where the vertical dimension has index 1 and the horizontal has index 2.

Having defined this new algorithm we need to prove that it solves the Cube Pruning problem in the monotonic case. The proof below shows that the Algorithm 3 (FCP1) outputs the exact $k$-best list, containing the top $k$ elements in $S$.

**Theorem 2.** Algorithm 3 solves the Cube Pruning Problem with monotonic search space.

**Proof.** By induction on the $k$-best list size:

- **Base case:**
  During the first iteration, $1$ is added to the $k$-best list. The monotonic search space assumption implies: $\phi(1) \leq \phi(v), \forall v \in S$. Thus when $|k$-best$| = 1$, the $k$-best list contains the best 1 element in $S$.

- **Induction step:**
  We assume that at the beginning of the $n$-th iteration of Algorithm 3 (line 4), the $k$-best list contains the best $n - 1$ elements. During the $n$-th iteration, an element $v$ is removed from $Q$ and is added in $k$-best. Let $B \equiv \{ u | \phi(u) \sqcap \phi(v), u \in S \}$, be the set of elements in $S$ that are “better” than $v$ according to the ordering function $\sqcap$. We distinguish three cases:

  1. $u \in B$ is a candidate in $Q$ ($u \in Q$): This event cannot occur, because in this case $u$ would have been chosen instead of $v$. 

2. \( u \in B \) has not been considered: Now we recursively apply \( A' \) to \( u \) and its ancestors, until we find the unique path that connects 1 with \( u \), \( \text{path}_u : < 1, w_1, \ldots, w_l, u > \).

From the monotonicity of the search space and definition of \( A' \) we have that: \( \phi(1) \odot \phi(w_1) \odot \cdots \odot \phi(w_l) \odot \phi(u) \). Therefore all elements in \( \text{path}_u \) are in \( B \). Knowing that 1 is already in \( k\)-best and \( u \) has not been inserted in \( Q \) and from the definition of Algorithm 3, we deduce that there is an element \( w_i \in \text{path}_u \) that has been inserted in \( Q \) but has not yet been moved into \( k\)-best. This implies that \( w_i \in B \) is a candidate in \( Q \), and this event cannot occur as shown in the case 1.

3. \( u \in B \) is in \( k\)-best: By exclusion this is the only possible case.

So we can state that all elements in \( B \) are in \( k\)-best: \( B \subseteq k\)-best.

Considering the loop invariant stated above: “at beginning of the \( x\)-th iteration, the \( k\)-best list contains already the top \( x-1 \) elements”, we know that \( k\)-best cannot contain elements that are “worse” than \( v \), therefore: \( B = k\)-best. We conclude that all the elements that are “better” than \( v \) are already in \( k\)-best, and there are no element “better” than \( v \) outside \( k\)-best. This proves that at the end of the \( x\)-th iteration, \( k\)-best contains all and only the best \( x \) elements.

\( \square \)

The proof of Theorem 2 also shows that FCP1 returns the \( k\)-best list already ordered.

As discussed above, FCP1 requires to specify an ordering of the coordinates of the search space. Anyway, the \( k\)-best list that is produced as output in a monotonic application is always the exact solution, and does not depend on the specific ordering applied. Instead, the set of elements that are in the queue at runtime differs according to the ordering of the coordinates chosen. For example, if Figure 2.4 (b) represented the execution of FCP1 on the same example but with reversed dimension order (horizontal dimension with index 1, and vertical with index 2), at the second step \( Q \) would contain the element with score 11, in the fourth step \( Q \) would not contain the element with score 13, and in the last
2.6. Faster Cube Pruning

step \( Q \) would contain the element with score 15 instead of the one with score 13. These differences in \( Q \) affect the decoding in the approximately monotonic search space. Thus, when FCP1 is applied to find an approximate solution for an approximately monotonic application, the chosen coordinates ordering affects the approximated \( k\)-best list that is returned as output.

2.6.2 Faster Cube Pruning 2

In this section we describe Faster Cube Pruning 2 (FCP2), the second algorithm we propose as improvement over SCP. To explain the intuition behind FCP2, consider the SCP runtime example depicted in Figure 2.4 (a). At the first round, 2 is inserted in the \( k\)-best, then 6 and 7 are added in the queue as candidates. In the second round, the smaller element from \( Q \) is 6. Therefore, 6 is moved into the \( k\)-best list. At this stage, SCP would add all the elements following 6 in the queue. Among the elements following 6 there is 11. Before adding 11 in \( Q \), we could notice that one of its ancestors, 7, is still in \( Q \). Thus, it is certain that at the next round 11 will not be selected to be moved in \( k\)-best. So its presence in \( Q \) is useless until 7 is selected. Anyway, after 7 is picked, the algorithm will try again to insert 11 in \( Q \). Therefore, we can avoid adding 11 at this step and just add 10 as element following 6.

In general, we can say that it is possible to avoid adding an element to \( Q \) until all its ancestors are in the \( k\)-best list. FCP2 uses the same \( F_v \) and \( A_v \) as the one defined for SCP. The difference is in the policy with which \( Q \) is updated. Figure 2.4 (c) pictures a running example for FCP2 for a bidimensional case.

The pseudocode of FCP2 is listed in Algorithm 4. FCP2 pseudocode is very similar to SCP pseudocode. The only difference is at line 8: a following element \( v' \) is inserted in \( Q \) only if all the ancestors of \( v' \) are already in the \( k\)-best list: \( A_{v'} \subseteq k\)-best. Obviously, FCP2 does not need to check if the neighbor \( v' \) is in the queue, because there is no risk that an element is inserted twice, since every element is inserted into \( Q \) only in the iteration where the last of its ancestors is inserted in \( k\)-best.

As we observed in Algorithm 3 (FCP1): avoiding the SCP’s line 8 test, allows us to avoid using an additional hash table. Unfortunately, for Algorithm 4 (FCP2) we replace that statement with: \( A_{v'} \subseteq \text{best-}K \), that requires the use of a data structure similar to the one used for SCP. Furthermore, the test must be repeated.
Algorithm 4  Faster Cube Pruning 2

1: function MainLoop ($\mathcal{X}$, $\odot$, $\oplus$, $\mathcal{L}$, $k$) : $k$-best
2: $Q \leftarrow \{1\}$;
3: $k$-best $\leftarrow$ empty-list;
4: while $|k$-best$| < k$ and $|Q| > 0$ do
5:   $v \leftarrow$ get-best($Q$, $\odot$);
6:   insert($k$-best, $\phi(v)$);
7:   for all $v' \in$ FollowersOf($v$) do
8:     if $A_{v'} \subseteq k$-best then
9:        insert($Q$, $v'$);
10:     end if
11:   end for
12: end while

13: function FollowersOf($v$): $\mathcal{F}_v$
14: $\mathcal{F}_v \leftarrow$ empty-set;
15: for $1 \leq i \leq |v|$ do
16:   $v^* \leftarrow v + b^i$
17:   if $v^*_i \leq |L_i|$ then
18:      insert($\mathcal{F}_v$, $v^*$);
19: end if
20: end for
a number of times proportional to \(|A_v|\). Fortunately this number is proportional to the number of dimensions in the search space, and that is constant or bounded in most cases. For example in Machine Translation it is preferable to use a grammar composed by binarized rules, in which case \(|A_v| \leq 2\).

Now we prove that Algorithm 4 (FCP2) solves the Cube Pruning Problem.

**Theorem 3.** Algorithm 4 solves the Cube Pruning Problem.

**Proof.** By induction on \(k\)-best size:

- **Base case:**
  
  During the first iteration, 1 is added to the \(k\)-best list. The monotonic search space assumption implies: \(\phi(1) \preceq \phi(v), \forall v \in S\). Thus when \(|k\text{-best}| = 1\), the \(k\)-best list contains the best 1 element in \(S\).

- **Induction step:**
  
  We assume that at the beginning of the \(x\)-th iteration of Algorithm 4 (line 4), the \(k\)-best list contains the best \(x - 1\) elements. During the \(x\)-th iteration, an element \(v\) is removed from \(Q\) and is added in \(k\)-best. Let \(B \equiv \{u | \phi(u) \preceq \phi(v), u \in S\}\), be the set of elements in \(S\) that are “better” than \(v\) according to the ordering function \(\preceq\). We distinguish three cases:

  1. \(u \in B\) is a candidate in \(Q\) (\(u \in Q\)): This event cannot occur because in this case \(u\) would have been chosen instead of \(v\).

  2. \(u \in B\) has not been considered (\(u \notin Q, u \notin k\text{-best}\)): Let us define function \(\gamma(\cdot): S \to \mathbb{N}\), so that \(\gamma(u) = \sum_{i=1}^{\lvert C \rvert} u_i\). In words, \(\gamma(\cdot)\) maps an element of the search space into the summation of its coordinates.

     From the monotonicity of the search space, we can imply that: \(A_u \subseteq B\), meaning that all the ancestors of \(u\) are “better” than \(v\).

     From \(A_u \subseteq B\) and \(w \notin Q, \forall w \in B\) (as shown in case 1), we can imply: \(A_u \cap Q = \emptyset\), meaning that no ancestor of \(u\) is in the queue of candidates \(Q\).

     The definition of the update policy of \(Q\) for Algorithm 4 requires that an element is inserted in \(Q\) if and only if the last of its ancestors is moved in the \(k\)-best list. Since \(u \notin Q\), we can deduce: \(\exists w \in A_u, w \notin Q\).
Chapter 2. Cube Pruning

$k$-best, meaning that, from the fact that $u$ is not in $Q$ yet, can be inferred that not all the ancestors of $u$ are in $k$-best.

We conclude that there must be at least one ancestor of $u$ that has not been considered yet: $\exists w_1 \in A_u, w_1 \notin Q, w_1 \notin k$-best.

If we apply the same reasoning recursively, we have that there exists a $w_2 \in A_{w_1}$ that has not been considered, and a $w_3 \in A_{w_2}$ that has not been considered, and so on.

Now we can find where this sequence leads using the $\gamma(\cdot)$ function: if $\gamma(u) = K$ then, from how $F_{w_1}$ is defined, we have that $\gamma(w_1) = K - 1$ and $\gamma(w_2) = K - 2$ and so on. We iterate until we reach the element $w_x$ such that $\gamma(w_x) = |L|$.

From the definitions of $w \in S$ and $\gamma(\cdot)$, we deduce that the only element $w_x$ for which $\gamma(w_x) = |L|$ is 1. Now we have reached an absurd because 1 cannot be in the chain of elements that have not been considered.

3. $u \in B$ is in $k$-best ($u \in k$-best): By exclusion this is the only possible case.

Having shown that every element in $B$ must have already been considered and cannot be in $Q$, we have to conclude that: all elements in $B$ have been already moved into $k$-best. Therefore $B \subseteq k$-best.

Considering the loop invariant stated above: “at beginning of the $x$-th iteration, the $k$-best list contains already the top $x - 1$ elements”, we know that $k$-best cannot contain elements that are “worst” than $v$, therefore: $B = k$-best. We conclude that all the elements that are “better” than $v$ are already in $k$-best, and there are no element “better” than $v$ outside $k$-best.

This proves that at the end of the $x$-th iteration, $k$-best contains all and only the best $x$ elements.

\[ \square \]

The proof of Theorem 3 also shows that FCP2 returns the $k$-best list already ordered.

Compared to Standard Cube Pruning, we expect FCP2 to be faster since it uses the minimal number of candidates in the queue. Compared to FCP1,
FCP2 inserts fewer elements in $Q$ but needs to use an additional data structure, such as a hash table. Given this analysis, we cannot be sure whether FCP1 or FCP2 is faster for a specific application. We will address this issue when testing these algorithm within a Hierarchical Machine Translation model in the empirical experiments in section 2.8.

2.6.3 Faster Cube Pruning 3

In this subsection, we introduce the Faster Cube Pruning 3 algorithm (FCP3). FCP3 can be considered an improvement over FCP2. FCP3 inserts the candidate elements in the same order and applies the same update policy as FCP2. FCP3 improves the efficiency of the test needed to select the candidates. To explain the intuition behind FCP3, consider Figure 2.4 (c). When an element $v = \langle i_{vertical}, i_{horizontal} \rangle$ is moved from the queue to the $k$-best list, the neighbor following along the horizontal dimension needs to be inserted in the queue iff the column $[i_{horizontal} + 1]$ does not already contain an element that is currently in the queue. Symmetrically, the neighbor following along the vertical dimension needs to be inserted in the queue iff the row $[i_{vertical} + 1]$ does not already contain an element that is currently in the queue. We can efficiently check if columns and rows already contain elements currently in the queue keeping a boolean value for each column and row. The test and update of these flags can be done in constant time. To generalize this algorithm to $N$ dimensions we just need to keep $N$ vectors of booleans. At every iteration, both FCP2 and FCP3 consider a neighbor for each dimension and for each neighbor make $N-1$ tests to decide to insert it in the queue. The difference between the two algorithms is that the tests in FCP3 are accesses to an array of booleans, while for FCP2 each test consists in a hash table access that is still constant time but has heavier computational cost to access and maintain, and requires the allocation of more memory. FCP1 still has the advantage over FCP3 that for FCP1 the set of new candidates to be inserted in the queue is a function of only $v$ and does not need to access an additional data structure, while FCP3 needs to access the array of booleans.

The pseudocode of FCP3 is listed in Algorithm 5. FCP3 pseudocode is very similar to FCP2 pseudocode. The main difference is that it uses a matrix of booleans to flag if a specific column or row already contains a candidate. In the pseudocode the matrix of booleans is represented with the “HasCandidate”
Algorithm 5  Faster Cube Pruning 3

1: function MainLoop ($\mathcal{X}$, $\otimes$, $\oplus$, $\mathcal{L}$, $k$) : $k$-best
2: $Q \leftarrow \{1\}$;
3: $k$-best $\leftarrow$ empty-list;
4: for all $i \in [1, |L|]$ do
5:     HasCandidate$[i][1] \leftarrow$ true;
6: end for
7: while $|k$-best$| < k$ and $|Q| > 0$ do
8:     $v$ $\leftarrow$ get-best($Q$, $\otimes$);
9:     insert($k$-best, $\phi(v)$);
10:    for all $i \in [1, |L|]$ do
11:        HasCandidate$[i][v_i] \leftarrow$ false;
12:    end for
13:    for all $v' \in$ FollowersOf($v$) do
14:        if HasCandidate$[i][v_i'] ==$ false $\forall i \in [1, |L|]$ then
15:            insert($Q$, $v'$);
16:        for all $i \in [1, |L|]$ do
17:            HasCandidate$[i][v_i'] \leftarrow$ true;
18:        end for
19:    end if
20: end for
21: end while

22: function FollowersOf($v$): $\mathcal{F}_v$
23: $\mathcal{F}_v$ $\leftarrow$ empty-set;
24: for $1 \leq i \leq |v|$ do
25:     $v^* \leftarrow v + b_i$
26:     if $v_i^* \leq |L_i|$ then
27:         insert($\mathcal{F}_v$, $v^*$);
28:     end if
29: end for
symbol. The matrix HasCandidate has 2 indices, the first index is used to choose the dimension (e.g. in a bidimensional case: 1 column, 2 row), the second index specifies the position within the selected dimension. Notice that this pseudocode is general for any number of dimensions. The HasCandidate matrix is initialized setting to true only the first position of all the dimensions at lines 4-6, since the only element in Q is 1. At lines 10-12, the coordinates corresponding to the element that has just been moved from Q to k-best are set to false. The test at line 12, checks if there is already a element in Q having any coordinate equal to the element candidate to be inserted in Q. At lines 14-15, the coordinates corresponding to the element that has inserted in Q are set to true. These lines replace the operations of query and updating the hash table that were implicit in the FCP2 pseudocode. Since FCP3 applies the same definitions of F and applies the same Q update policy than FCP2, then the proof of correctness produced for FCP2 holds for FCP3. Figure 2.4 (c) is an example of execution of the FCP3 as well.

2.7 Linear Time Cube Pruning

So far we have introduced a set of Fast Cube Pruning algorithms that solve the monotonic CP problem exactly. These algorithms improve the efficiency of the Standard Cube pruning algorithm under different aspects. Despite that, the overall complexity of the FCP algorithms is still $O(k \log k)$. In this section we address the issue of finding a solution to the monotonic CP problem that runs with complexity to $O(k)$. Unfortunately, there seems to be no way to achieve an algorithm that is asymptotically faster than $O(k \log k)$ and that solves exactly the monotonic CP problem. Nonetheless, given the impossibility of finding an exact linear time solution, we allow relaxations on the original monotonic CP problem. Thus, our goal is to find the most accurate linear time approximate algorithm solving the CP problem. In other words, we want to know what is the best that can be done in linear time, even allowing some degree of approximation.

For the sake of simplicity, we are going to constrain the discussion to the bidimensional search space case. The extension to a search space with an arbitrary number of dimensions can be generated by applying the extension technique described in section 2.4.6
Chapter 2. Cube Pruning

Figure 2.5: Space of the results obtained by summing an element of $L_1 = \langle 1, 5, 9, 13 \rangle$ to an element of $L_2 = \langle 1, 6, 8, 15 \rangle$.

2.7.1 Linear Cube Pruning Intuition

In this section we give an intuitive explanation of how the Cube Pruning algorithm can be approximated with a linear time solution and we sketch the incremental process that led to its definition, by following a sequence of less stringent relaxations of the CP problem.

Can the problem of finding the best $k$ elements in the CP search space be solved with an asymptotically faster algorithm? The CP algorithm has complexity of $O(n \log n)$. The Faster CP algorithms optimize different aspects of the standard CP algorithm, but still have overall $O(n \log n)$ complexity. Let us consider the sample CP search space represented in Figure 2.5 and wonder if a solution can be found in linear time. The number of iterations is bound to $n$ unless we can insert more than one element in the out list per iteration. And finding the best element to be moved in the out list requires the $\log n$ factor unless we find a way to retrieve it in constant time. It appears that $O(n \log n)$ is a lower bound for an exact solution.

Thus we change our question. What is the best we can do in linear time? In other words, assuming that we are happy with an approximate algorithm, which linear time algorithm returns a list that is as close as possible to the correct one?

An extremely simple linear time algorithm would be to collect the results by scanning the search space following diagonals of 45 degrees, starting from the top left corner and continuing each time with the more external diagonal. An example of this algorithm is depicted in Figure 2.6. This algorithm would return the right solution if we constraint the input lists to have constant and equal
2.7. Linear Time Cube Pruning

Figure 2.6: Space of the results obtained by summing two lists with constant and equal derivatives. The derivative is $\Delta = 3$. The input lists are $L_1 = \langle 1, 4, 7, 10 \rangle$ to an element of $L_2 = \langle 0, 3, 6, 9 \rangle$.

Figure 2.7: Space of the results obtained by summing two lists with constant but different derivatives. The derivatives are $\Delta_1 = 6$ and $\Delta_2 = 3$. The input lists are $L_1 = \langle 1, 7, 13, 19 \rangle$ to an element of $L_2 = \langle 0, 3, 6, 9 \rangle$.

derivatives. This is the case of the example in Figure 2.6. The input lists are not the same but have same derivative $\Delta = 3$. Thus the elements on the diagonals are all equivalent, and collecting them following the diagonals returns the correct out list.

What if we have as input two lists with constant but different derivative? An example of this is the case represented in Figure 2.7. The first list has constant derivative $\Delta_1 = 6$ and the second list has constant derivative $\Delta_2 = 3$. In that case we can retrieve the correct out list in constant time following diagonal with a pendence proportional to the ratio of the two derivatives. More precisely the diagonal inclination will be equal to $\arctan(\Delta_1/\Delta_2)$. In our example the inclination is 75 degrees. Following this inclination the elements can be collected in the right order as shown in Figure 2.6
Now we wonder if there is a linear time algorithm returning the correct output list if only one list has constant slope, while the other list can be any. It turns out that such algorithm exists. In the rest of the chapter we will define it in details and provide a formal prove of its correctness. Now we give an intuitive introduction of the principle that makes this algorithm possible.

Assume we are given as input two lists of which only one has constant derivative $\Delta$. If we place the list with constant derivative on the horizontal dimension, then the columns of the search space will be all equal to the input list without constant derivative summed to a constant value. Where the constant value is a multiple of $\Delta$. Assume we are scanning the search space generated by these two lists and we find the elements 7, 8, 9 in the cells shown in Figure 2.8. Then we can infer that if we take the pattern of the path that connects these 3 cells and we shift it along the horizontal dimension, the 3 cells that we find are guaranteed to respect the same ordering. The same reasoning can be applied for any other sequence pattern (e.g. if we swap 8 with 9, we obtain a different pattern, that is still valid after any horizontal shift). Thus, if we store the pattern of the previously scanned cells, we can find the next element in constant time by following a shifted version of the recorded pattern. This is the basic idea on which the Linear Time Cube Pruning algorithm is based.

2.7.2 Naive Cube Pruning in Linear Time

This section is introductory for the Linear Cube Pruning algorithm. We describe a relaxation of the monotonic Cube Pruning problem for which a naive linear
time solution can be easily found. This Naive Cube Pruning algorithm serves
as an introduction to the intuition on which we base the Linear Cube Pruning
algorithm (LCP). LCP is the algorithm that we will propose as our best linear
time approximation to the CP problem.

Let us define the property **constant slope** for a sorted list $L_n \equiv \langle x_{n,1}, \ldots, x_{n,|L_n|} \rangle$.

We say that $L_i$ has constant slope if there exists $\Delta \in \mathcal{X}$ such that:

$$x_{n,i} \oplus \Delta = x_{n,i+1}$$

for all $i$ in $1 \leq i < |L_n|$.

Now we can state the first relaxation of the monotonic CP problem:

**Problem 4. Cube Pruning Problem in Monotonic Search Space with
equal and constant slopes:**

This problem is defined by relaxing the Problem 1 (monotonic CP) by adding
the following assumptions on the input:

- Bidimensional search space: $\mathcal{L} \equiv \{L_1, L_2\}$, meaning that the input set
  contains only two sorted lists.
- Both input lists have constants slope: meaning that $L_1$ has constant slope
  $\Delta_1$ and $L_2$ has constant slope $\Delta_2$.
- The input lists have equal constant slope: $\Delta_1 = \Delta_2$.

As we mentioned, the assumption on the bidimensional search space is added
just to facilitate the discussion and can be removed using the extension technique
described in section 2.4.6. The other two assumptions constrain the search space
to be a flat surface. In these conditions, the $k$-best elements can be found in linear
time with a naive algorithm that just visits the search space following diagonals
with $45^\circ$ pendence.

We refer to this algorithm as Naive Cube Pruning (NCP). The NCP
pseudocode is reported in Algorithm 6. At line 2, the $k$-best is initialized to
the empty list. At line 3, $v$ is initialized to $1$. $v \equiv \langle v_1, v_2 \rangle$ is used as iterator to
follow the diagonals. At line 4 the main loop starts, it terminates when $k$-best
contains $k$ elements. At line 5 the $\phi(\cdot)$ value correspondent to $v$ is added to
Algorithm 6  Naive Cube Pruning

1: function MainLoop \((\mathcal{X}, \oplus, \odot, L_1, L_2, k) : k\)-best
2: \(k\)-best \(\leftarrow\) empty-list;
3: \(v \leftarrow \langle 1, 1 \rangle\);
4: \(\textbf{while} \ |k\text{-best}| < k \textbf{ do}
5: \(\quad\) insert\((k\text{-best}, \phi(v))\);
6: \(\quad v \leftarrow \langle v_1 + 1, v_2 - 1 \rangle\);
7: \(\quad\) if \(v_1 > |L_1|\) or \(v_2 < 1\) then
8: \(\quad\quad\) \textbf{break};
9: \(\quad\) else if \(v_1 + v_2 \leq |L_2|\) then
10: \(\quad\quad v \leftarrow \langle 1, v_1 + v_2 \rangle\);
11: \(\quad\) else
12: \(\quad\quad v \leftarrow \langle 1 + v_1 + v_2 - |L_2|, |L_2| \rangle\);
13: \(\quad\) \textbf{end if}
14: \(\textbf{end if}
15: \(\textbf{end while}

\(k\)-best. At line 6, \(v\) is moved to the next position following the diagonal. The test at line 7 checks if the current diagonal is finished. If this is the case three cases are distinguished: (lines 8-9) if the diagonal that just finished was the last one intersecting the search space, then the loop ends; (lines 10-11) if the next diagonal intersects the first row, then the iterator is placed at the start of the next diagonal from the first row, (lines 12-13) in the remaining case, the next diagonal intersects the search space but begins with an element of the last column, then the iterator is placed accordingly.

Notice that NCP makes no use of the queue of candidates \(Q\). Thus there is no need of inserting elements in order and the \(\log k\) factor is removed from the asymptotic complexity. NCP runs in linear time. To formally prove that Algorithm 6 solves exactly Problem 4 is trivial and we leave it to the reader. Notice that all the elements on the diagonals have the same value, therefore it does not matter in which direction the elements are collected.

If we apply NCP to solve the monotonic CP problem, it will return a \(k\)-best list that is not guaranteed to be correct. Having defined NCP, we would like to find a linear time algorithm that improves over NCP. In the next section we formalize a more stringent relaxation of the monotonic CP problem by removing some of the relaxing assumptions added for Problem 4. Then, we define a better
2.7. Linear Time Cube Pruning

A linear time algorithm, that solves exactly the more stringent relaxation of the monotonic CP problem.

2.7.3 Linear Cube Pruning algorithm

As discussed above, there seems to be no way to achieve an algorithm solving exactly the monotonic CP problem that is asymptotically faster than $O(k \log k)$, even by exploiting the restriction that the search space can be interpreted as a set of ordered lists all related by some shifts, as described in Section 2.4.5. Nonetheless, in this section we use the above ideas to develop an approximate CP algorithm running in linear time.

Following the methodology that led to the definition of the Naive CP algorithm, we start by defining a relaxation of the monotonic CP problem (Problem 1).

Problem 5. Cube Pruning Problem in Monotonic Search Space with one constant slope dimension:

This problem is defined by extending the Problem 1 by adding the following assumptions on the input:

- Bidimensional search space: $\mathcal{L} \equiv \{L_1, L_2\}$, meaning that the input set contains only two sorted lists.
- Only one input list has constant slope: $L_2$ has constant slope $\Delta$.

As we discussed, the assumption on the bidimensional search space is added just to ease the discussion and can be removed using the extension technique described in section 2.4.6. Therefore there is only one additional assumption constraining one dimension to have constant slope.

This relaxation is less stringent than the relaxation that led to the NCP algorithm (Problem 4). Notice that the assumptions added by Problem 5 are a subset of the assumptions added by Problem 4. Thus, Problem 4 can be considered as a relaxation of Problem 5. And therefore, all the algorithms that are guaranteed to be exact for Problem 5 are guaranteed to be exact for Problem 4 too. Thus, an algorithm that is exact for Problem 5 has lower chances to make mistakes than
the NCP algorithm, when applied as approximate solution of the monotonic CP
problem.

The algorithm we propose as an exact linear time solution to Problem 5 is
listed in Algorithm 7. We refer to this algorithm as Linear Cube Pruning (LCP).

Before starting to explain the LCP algorithm, let us define the data struc-
tures and terminology that are going to be employed. LCP makes use of a data
structure known as circular list, $C$. A circular list is defined by a list of nodes
$\langle node_1, node_2, \ldots, node_{|C|} \rangle$. Each node stores two things: a link to the next node;
and a generic object. The link connects $node_i$ to $node_{i+1}$ for all $1 \leq i < |C|$, and
connects $node_{|C|}$ back to $node_1$. For our application, the object stored in each
node is a bidimensional vector. As we will discuss in detail later, the vector rep-
resents a relative position in the bidimensional search space, and the circular list
represents a relative path in the search space.

To iterate and operate on the circular list we use a circular list iterator, $C$-
iterator. The iterator points to a node of the circular list. The iterator is provided
with methods to read and modify the list:

- The method $C$-iterator.current() returns the vector stored in the currently
pointed node.
- The method $C$-iterator.next() moves the iterator to the next node, and
returns its vector.
- The method $C$-iterator.insert($\langle i, j \rangle$) inserts a new node before the one cur-
currently pointed, to the new node stores the vector passed as argument.

To ease the explanation of the LCP algorithm, we will make references to the
running example reported in Figure 2.9. The two input lists are $L_1 = \langle 1, 6, 8, 13 \rangle$,
$L_2 = \langle 1, 4, 7, 10 \rangle$. $L_2$ has constant slope with $\Delta = 3$. Each of the pictures in the
sequence represents the state of the algorithm when the test at line 9 is executed.

The LCP algorithm starts by initializing the $k$-best list to a list containing the
$\phi(\cdot)$ value of $1 = \langle 1, 1 \rangle$. For the monotonicity of the search space $1$ is guaranteed
to be the best element in the search space. In Figure 2.9, the bidimensional
search space is represented with a $4 \times 4$ matrix. The cells with white background
represent the values that have been inserted in the $k$-best. At the first iteration
(Figure 2.9 a), only $\phi(\langle 1, 1 \rangle) = 2$ is in the $k$-best. At line 3, the circular list, $C$, is
initialized with a single node, containing the vector $\langle 1, 1 \rangle$. This initial state of $C$
Algorithm 7  Linear Cube Pruning

1: function MainLoop ($\mathcal{X}$, $\otimes$, $\oplus$, $L_1$, $L_2$, $k$) : $k$-best
2: $k$-best $\leftarrow \{\phi(\langle 1, 1 \rangle)\}$;
3: $C \leftarrow \text{CircularList}(\langle 1, 1 \rangle)$;
4: $C$-iterator $\leftarrow C$.begin();
5: referColumn $\leftarrow 1$;
6: $v_{\text{follow}} \leftarrow \langle 1, 2 \rangle$;
7: $v_{\text{deviate}} \leftarrow \langle 2, 1 \rangle$;
8: while $|k$-best$| < k$ do
9:   if $\phi(v_{\text{follow}}) \otimes \phi(v_{\text{deviate}})$ then
10:      insert($k$-best, $\phi(v_{\text{follow}})$);
11:     if $C$-iterator.current() == $\langle 1, 1 \rangle$ then
12:        referColumn++;
13:   end if
14:   $\langle i, j \rangle \leftarrow C$-iterator.next();
15:   $v_{\text{follow}} \leftarrow \langle i, j + \text{referColumn} \rangle$;
16: else
17:      insert($k$-best, $\phi(v_{\text{deviate}})$);
18:      $C$-iterator.insert($\langle v_{1}^{\text{deviate}}, v_{2}^{\text{deviate}} - \text{referColumn} \rangle$);
19:      $v_{\text{deviate}} \leftarrow \langle v_{1}^{\text{deviate}} + 1, v_{2}^{\text{deviate}} \rangle$;
20: end if
21: end while

can be represented as a single node having an arc looping on itself, as depicted in
Figure 2.9 a. At line 4, the circular list iterator, $C$-iterator, is initialized to point
to the only node in $C$. The $C$-iterator is represented by a blue arrow in Figure
2.9. At line 5, the variable referColumn is initialized to 1. referColumn stores
the index of the column used as reference for the relative path encoded by $C$.
Only the column index is needed to be stored as reference and not a row index,
because the relative path is affected only by horizontal displacements. In Figure
2.9, a red arrow is used to point to the reference column.

The LCP algorithm uses two vectors, $v_{\text{follow}}$ and $v_{\text{deviate}}$, to point to the two
elements candidate to be inserted next in $k$-best. $v_{\text{follow}}$ represents the element
to be inserted if the algorithm chooses to follow the path encoded by $C$. $v_{\text{deviate}}$
represents the element to be inserted if the algorithm chooses to deviate from $C$.
$v_{\text{follow}}$ is initialized to $\langle 1, 2 \rangle$ at line 6. $v_{\text{deviate}}$ is initialized to $\langle 2, 1 \rangle$ at line 7. In
Figure 2.9, the value in the shaded cell in the first column is $v_{\text{deviate}}$, while the
value in the other shaded cell is $v_{\text{follow}}$. At line 8 the main loop starts. It ends
Figure 2.9: A running example for the Linear Cube Pruning Algorithm.

when the $k$-best list contains $k$ elements. At line 9, LCP chooses if it is the case
to follow the path or deviate from it. If the $\phi(\cdot)$ value associated to $v_{\text{follow}}$ is
better than $v_{\text{deviate}}$ according to the ordering function $\preceq$, then lines 10-15 are
executed, otherwise lines 16-19 are executed.

At line 10, $\phi(v_{\text{follow}})$ is inserted in $k$-best. At line 11, the algorithm tests if
the $C$-iterator is currently pointing to $(1,1)$. If this is the case, it means that a
complete iteration over the path encoded in $C$ has just been completed. Therefore,
the algorithm needs to displace the relative path one column to the right. This
is done simply by incrementing the reference column at line 12. At line 14,
the iterator is moved to the next node. The value of the next node is stored in
2.7. Linear Time Cube Pruning

At line 15, \( v^{\text{follow}} \) is updated to \( \langle i, j \rangle \) displaced to the right by the value of referColumn.

At line 17, \( \phi(v^{\text{deviate}}) \) is inserted in \( k\text{-best} \). After a deviation, the relative path is updated to be able to produce the same deviation in future iterations. This is done at line 18, by adding to \( C \) a node storing the value of the current \( v^{\text{deviate}} \) minus the current value of the reference column. At line 19, \( v^{\text{deviate}} \) is updated by moving it down to the next row. Notice that \( v_2^{\text{deviate}} \) is initialized to 1, and is never incremented. Thus \( v^{\text{deviate}} \) is always on the first column. Every time the current \( v^{\text{deviate}} \) is added to \( k\text{-best} \), the next \( v^{\text{deviate}} \) is the following element on the first column. Therefore, \( v_2^{\text{deviate}} \) is always equal to 1, and we can substitute all the occurrences of \( v_2^{\text{deviate}} \) with the constant 1.

Now we prove that LCP (Algorithm 7) solves exactly the Cube Pruning Problem in monotonic search space with one constant slope dimension (Problem 5).

**Theorem 4.** Algorithm 7 solves Problem 5.

**Proof.** By induction on \( k\text{-best} \) size:

- **Base case:**
  At the beginning of the first iteration, \( k\text{-best} \) contains only 1. The monotonic search space assumption implies: \( \phi(1) \subseteq \phi(v), \forall v \in S \). Thus when \( |k\text{-best}| = 1 \), the \( k\text{-best} \) list contains the best 1 element in \( S \).

- **Induction step:**
  We assume that at the beginning of the \( x \)-th iteration of Algorithm 7, the \( k\text{-best} \) list contains the values of the best \( x \) elements. During the \( x \)-th iteration, an element \( v \) is added in the \( k\text{-best} \) list.

Now we define a partition of the search space, \( S \), into 5 disjoint subsets. At the beginning of any iteration, the search space can be partitioned in the following sets:

1. \( K \), the set of elements whose values are already in the \( k\text{-best} \) list.
2. \( \{v^{\text{follow}}\} \), the set containing only the \( v^{\text{follow}} \) element.
3. \( \{v^{\text{deviate}}\} \), the set containing only the \( v^{\text{deviate}} \) element.
4. \( T \), the set containing the remaining elements of the top portion of the search space. We define the top portion of \( S \) as:

\[
S^{\text{top}} \equiv \{(i,j) \mid i < v^{\text{deviate}}_1, (i,j) \in S\}
\]  

(2.18)

In words, \( S^{\text{top}} \) contains all the elements that are above the row containing \( v^{\text{deviate}} \). Then we can formally define \( T \) as:

\[
T \equiv S^{\text{top}} \setminus \{K \cup v^{\text{follow}}\}
\]  

(2.19)

In words, \( T \) contains all the elements of the top portion of the search space excluded for the elements in \( K \) and \( v^{\text{follow}} \). Notice that \( v^{\text{deviate}} \) is excluded from \( S^{\text{top}} \) by definition.

5. \( B \), the set containing the remaining elements of the bottom portion of the search space. We define the bottom portion of \( S \) as:

\[
S^{\text{bottom}} \equiv \{(i,j) \mid i \geq v^{\text{deviate}}_1, (i,j) \in S\}
\]  

(2.20)

In words, \( S^{\text{bottom}} \) contains all the elements that are in the same row containing \( v^{\text{deviate}} \) or below. Then we can formally define \( B \) as:

\[
B \equiv S^{\text{bottom}} \setminus \{v^{\text{deviate}}\}
\]  

(2.21)

In words, \( B \) contains all the elements of the bottom portion of the search space excluded for \( v^{\text{deviate}} \). Notice that \( K \) and \( v^{\text{follow}} \) are excluded from \( S^{\text{bottom}} \) by definition.

Notice that:

\[
S = K \cup v^{\text{follow}} \cup v^{\text{deviate}} \cup T \cup B
\]  

(2.22)

And the 5 sets are disjoints. Now we investigate the relations between these sets:

- Any element in \( K \) is better than any element in any of the other 4 subset. Formally:

\[
\phi(v) \sqsubseteq \phi(u) \quad \forall \ v \in K, \ u \in \{v^{\text{follow}} \cup v^{\text{deviate}} \cup T \cup B\}
\]  

(2.23)
2.7. Linear Time Cube Pruning

This is a consequence of the induction step assumption: “at the beginning of the x-th iteration k-best contains the values of the best x elements in S”.

- \( v^{\text{deviate}} \) is better than any element in \( B \). Formally:

\[
\phi(v^{\text{deviate}}) \preceq \phi(u) \quad \forall \ u \in B
\] (2.24)

This can be easily proven considering the monotonicity of the search space and the definition of \( B \). All the elements in \( B \) are below or at the right of \( v^{\text{deviate}} \). Thus for the monotonicity of the search space their value is worst than \( \phi(v^{\text{deviate}}) \).

- \( v^{\text{follow}} \) is better than any element in \( T \). Formally:

\[
\phi(v^{\text{follow}}) \preceq \phi(u) \quad \forall \ u \in T
\] (2.25)

This can be proven by considering the sequence of elements generated by looping over the path encoded in \( C \). Let us consider a generic element \( u \equiv \langle u_1, u_2 \rangle \in T \). Let us name the sequence of elements generated after the current \( v^{\text{follow}} \) by iterating over \( C \) and increasing the reference column every time the origin node is reached (as described in Algorithm 7): \( [v^{\text{follow}}, w^1, w^2, \ldots] \). From how \( C \) is constructed and from the constant derivative property we have that:

\[
\phi(v^{\text{follow}}) \preceq \phi(w^1) \preceq \phi(w^2) \preceq \cdots
\] (2.26)

Let us name \( w^i = \langle w^i_1, w^i_2 \rangle \) the first element in the sequence so that \( w^i_1 = u_1 \). In words \( w^i \) is the first element in the sequence that appears on the same row as \( u_1 \). Notice that this element must exist since \( w^i_1 \prec v^{\text{deviate}}_1 \). And notice that \( \langle w^i_1, w^i_2 - 1 \rangle \) must be already in \( K \) since it occurs in the path before \( v^{\text{follow}} \). This implies that \( w^i_2 \leq u_2 \). And for the monotonicity of the search space we can deduce that \( \phi(w^i) \preceq \phi(u) \). It follows that \( \phi(v^{\text{follow}}) \preceq \phi(u) \). This proves that \( v^{\text{follow}} \) is better than any element in \( T \).

Now we distinguish 2 cases:
1. if \( \phi(v_{\text{follow}}) \preceq \phi(v_{\text{deviate}}) \):

\( \phi(v_{\text{follow}}) \) is going to be inserted in \( k\text{-best} \). Using the properties we have discussed above:

\[
\phi(v_{\text{follow}}) \preceq \phi(u) \quad \forall \ u \in \{v_{\text{deviate}} \cup B \cup T\}
\]  

(2.27)

This proves that in the case \( \phi(v_{\text{follow}}) \) is added in the \( k\text{-best} \), then \( k\text{-best} \) contains the values of the best \( x+1 \) elements at the end of the \( x\)-th iteration.

2. Otherwise:

\( \phi(v_{\text{deviate}}) \) is going to be inserted in \( k\text{-best} \). Using the properties we have discussed above:

\[
\phi(v_{\text{follow}}) \preceq \phi(u) \quad \forall \ u \in \{v_{\text{deviate}} \cup B \cup T\}
\]  

(2.28)

This proves that in the case \( \phi(v_{\text{deviate}}) \) is added in the \( k\text{-best} \), then \( k\text{-best} \) contains the values of the best \( x+1 \) elements at the end of the \( x\)-th iteration.

This proves the induction step showing that at the end of the \( x\)-th iteration, \( k\text{-best} \) contains the values of the \( x+1 \) best elements.

\[\square\]

The proof of Theorem 4 also shows that LCP returns the \( k\text{-best} \) list already ordered.

**LCP as K-way merge problem**

Problem 5 can be mapped to a subclass of K-way merge problems, as we showed for Problem 1 in section 2.4.5. By mapping an instance of Problem 5 we obtain a K-way merge problem in which all the input lists are related by K shifts of value \( \Delta \). Formally, if \( \langle L_1^\text{mrg}, L_2^\text{mrg}, \cdots, L_K^\text{mrg} \rangle \) are the \( K \) input lists of the K-way merge problem, we have that:

\[
L_i^\text{mrg} \oplus \Delta = L_{i+1}^\text{mrg} \quad \forall \ 1 \leq i < K
\]  

(2.29)
Any instance of the K-way merge problem having this additional constraint can be solved in linear time using the LCP algorithm.

Extension to a Multidimensional Search Space

The LCP algorithm can be extended to work with any number of dimensions by applying the extension technique described in section 2.4.6. The extended LCP is guaranteed to return the exact solution if all the dimensions but one have constant slope. Formally, if \( \langle L_1, L_2, \ldots, L_N \rangle \) are the \( N \) input lists to the constrained CP problem, then LCP returns the exact solution if at least \( N - 1 \) lists have constant slope. The other lists must have constant slope, but are not required to have the same slope, for example each list can have a different slope. This can be shown easily: we rename the lists so that \( L_1 \) has not constant slope, while \( L_2, \ldots, L_N \) have constant slope. The extension technique can be applied by iteratively calling the bidimensional LCP on a pair of lists of which only one have constant slope. In practice, we can start executing LCP on the pair \( (L_1, L_2) \). This will return the list, \( L'_2 \), that is not guaranteed to have constant slope. Then we iterate calling LCP on the pair \( (L'_2, L_3) \). We get the list \( L'_3 \) and iterate until we get the final \( k \)-best. The extended LCP algorithm is still linear with respect \( k \). The only additional factor is proportional to \( N \) as explained in Section 2.4.6.

As discussed in section 2.4.7, The SCP algorithm returns the exact solutions for CP problems with monotonic search space (Problem 1). When SCP is applied as heuristic to solve the CP in approximately monotonic search space, SCP is not guaranteed to return the exact solution. Similarly, LCP can be applied as linear time heuristic algorithm to solve CP problems in approximately monotonic search space. We will test the application of the proposed CP algorithm on a real world application requiring to solve the CP problem in approximately monotonic search space in the following section.
2.8 Experimental Comparison on Approximately Monotonic CP

So far, we presented a set of algorithms solving exactly the monotonic CP problem and some of its relaxations. As discussed in section 2.4.7, to find an exact solution to the approximately monotonic CP problem, we cannot avoid to explore the entire search space. The approach that has been shown to provide a good balance between accuracy and speed is to solve the approximately monotonic CP problem using SCP as heuristic algorithm. In this section we apply the algorithms we presented to generate a heuristic solution to the approximately monotonic CP problem, and compare them with SCP. We will empirically show that the lower complexity leads to a shorter execution time with no significant variation in accuracy. The task we are going to test on is Hierarchical Machine Translation (HMT). HMT with Language Model feature is a well-known problem requiring to solve CP in an approximately monotonic search space.

We implemented the proposed algorithms on top of, cdec (Dyer et al. 2010), a widely-used hierarchical Machine Translation system that includes an implementation of a CKY-like bottom-up decoder for translation hypergraphs, that allows to execute beam search with a standard implementation of the SCP algorithm. Implemented in C++, it is known to be one of the fastest decoders.

For the sake of comparability, we experiment on the widely used NIST 2003 Chinese-English parallel corpus. The training corpus contains 239k sentence pairs with 6.9M Chinese words and 8.9M English words. The accuracy of the decoding algorithms is compared on the NIST-03 test set, which contains 919 sentence pairs.

A hierarchical phrase-based translation grammar was extracted using a suffix array rule extractor (Lopez 2007). The model was tuned using Minimum Error Rate Training (MERT) Och (2003). The workstation used has Intel Core2 Duo CPU at 2.66 GHz with 4M of cache, 3.5 GB RAM, and is running Linux kernel version 2.6.24 and gcc version 4.2.4. The features used are basic lexical features, word penalty and a 3-gram Language Model (Heafield 2011).
2.8. Experimental Comparison on Approximately Monotonic CP

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>BLEU</th>
<th>Average Score</th>
<th>Relative Probability Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCP</td>
<td>32.189</td>
<td>-119.5614</td>
<td>-119.5614</td>
</tr>
<tr>
<td>FCP1</td>
<td>32.223</td>
<td>-119.5734</td>
<td>-119.5734</td>
</tr>
<tr>
<td>FCP2</td>
<td>32.287</td>
<td>-119.6050</td>
<td>-119.6050</td>
</tr>
<tr>
<td>FCP3</td>
<td>32.287</td>
<td>-119.6050</td>
<td>-119.6050</td>
</tr>
<tr>
<td>NCP</td>
<td>31.844</td>
<td>-120.0304</td>
<td>-120.0304</td>
</tr>
<tr>
<td>LCP</td>
<td>32.220</td>
<td>-119.6338</td>
<td>-119.6338</td>
</tr>
</tbody>
</table>

Table 2.1: Evaluation and comparison of the accuracy with beam size = 30. For each of the algorithms presented the **BLEU** score (second column), the average internal search score (third column), and variation in terms of probability with respect to SCP (fourth column) are reported.

2.8.1 Accuracy Comparison

We begin the accuracy evaluation by comparing all the presented Cube Pruning algorithms setting the beam size to 30. Notice that this implies the $k = 30$ for the CP sub problems. In Table 2.1 we report the measures collected. One of the most used accuracy metric for Machine Translation systems is the **BLEU** score (Papineni et al. (2002)). In the second column of Table 2.1 we report the **BLEU** score achieved by different algorithms. NCP has a noticeably lower score, while all the other algorithms reach approximately the same **BLEU** score. The **BLEU** score variations for these algorithms cannot be considered significant. Since we compare decoding algorithms on the same search space, the best way to compare the accuracy is in terms of search score. The search score is the score on which the decoder bases the bottom-up search and the pruning actions. For each algorithm we compute the average score of the best translation found for the test sentences. The search score can be interpreted as the logarithm of the probability mass assigned to each translation. In the third column of Table 2.1 are reported the averages of the search score for each algorithm. We notice that the search score values are directly proportional to the amount of search space explored by each algorithm. SCP is the algorithm exploring the biggest portion of the search space and it reach the best search score. The FCPs algorithms prune more than SCP, and get a score that is slightly lower than SCP. Among the FCPs algorithms, FCP1 is the one getting the best score, and is also the one exploring a bigger portion of the search space among the FCPs algorithms. FCP2 and FCP3 have the same accuracy since they insert the candidate elements in the same order and
apply the same update policy as explained in section 2.6.3. LCP is performing slightly worst. NCP performs particularly badly.

It is not easy to interpreter these scores as they appear. To give a more meaningful comparison, we compute the variation with respect to the SCP score in terms of probability. Given an average search score $X$, the probability mass associated to it is: $C \cdot e^X$, where $C$ is a constant factor. The probability variation with respect to SCP is:

$$\frac{C \cdot e^X - C \cdot e^{X_{SCP}}}{C \cdot e^{X_{SCP}}} = \frac{e^X - e^{X_{SCP}}}{e^{X_{SCP}}}$$ (2.30)

Where $X_{SCP}$ is the score associated to SCP.

In the fourth column of Table 2.1 we report the variations in terms of probability in comparison to SCP. We can see that the FCP1 performs well, losing only $\approx 1\%$. Other algorithms have a moderate loss $< 7\%$. Except for NCP, that loses more than one third of the probability mass. This result proves that LCP performs much better than a Naive linear time algorithm, and actually its performance are close to the $O(n \log n)$ algorithms.

For the first series of experiments the beam size was set to 30. Beams of this magnitude are typical for systems that focus on optimizing the speed. Systems optimizing accuracy generally use larger values for the beam, in the order of hundreds. We repeat the previous experiments for a beam size of 300, and report the results in Table 2.2. The ordering of the algorithms is similar to the one obtained in the previous set of experiments. The main difference is that the gaps between the algorithms are smaller. This can be justified observing that by
increasing the beam size, all the CP algorithms converge to the exhaustive search. Also notice that FCP1 performs slightly better than SCP. This is due to the fact that with this beam size the two algorithms performance are really close, and a perturbation in an intermediate pruning step may lead FCP1 to prune out a sub-tree that have better partial score, but leads to complete derivations with better global score. Thus the $+0.1101\%$ increase in average probability mass obtained by FCP1 is to be considered not significant, and FCP1 accuracy with $k = 300$ is to be considered at the same level as SCP’s accuracy.

Now we wish to investigate in more details how the accuracy changes with the beam size. We compute the average score of the best translation found for the test sentences for all the algorithms at different beam sizes in the range $[1, 10000]$. We can safely assume that all these algorithms have a lower bound in loss in accuracy, because for larger beam size all the algorithms tend to converge to exhaustive search. Thus the score-variation curves will tend to 0 without significant variations. The collected measures are plotted in Figure 2.10. The blue curve plots the FCP accuracy variation. The blue curve stays close to the $x$ axis. This means that FCP performs really close to the SCP accuracy. In many
points FCP accuracy is slightly better than SCP, and the blue curve cross the $x$ axis. As we discussed above in more details, this is evidence that FCP accuracy is not significantly different from SCP, and noise can cause FCP to perform better than SCP in few instances. In general, FCP1 variation has a lower bound of $\approx -2\%$.

The violet curve plots the FCP2 and FCP3 accuracy variation. As discussed above, FCP2 and FCP3 explore exactly the same portion of the search space thus their accuracy is the same, they only differ in the speed comparison as we will see later. We notice that the violet curve is generally below the blue, but it is still close to the $x$ axis. As we expected, FCP2/3 perform a little bit worst than SCP and FCP1, since FCP2/3 explore a smaller portion of the search space. From beam size = 1000 we notice that FCP2/3 occasionally performs better than SCP and FCP1. This shows that, for beams bigger than 1000, FCP2/3 are to be considered converged to SCP and FCP1, and the quality of its output should not considered significantly different. FCP2/3 variation has a lower bound of $\approx -3.5\%$.

The red curve plots the LCP accuracy variation. LCP accuracy is clearly below the $O(n \log n)$ algorithms, and does not show signs of convergence before beam = 10000. Anyway the accuracy is reasonably good and LCP variation has a lower bound of $\approx 7\%$ that is touched with a beam in the range $[100, 1000]$. After beam 1000, LCP starts to slowly move toward convergence. LCP accuracy is to be considered pretty good for a linear time algorithm, and possibly it is the best that can be achieved in linear time. The gain over the Naive linear time CP algorithm is evident. NCP accuracy variation is plotted with a green line. It reaches the peak of the relative loss with beam in the range $[10, 100]$, losing $\approx 40\%$ of the probability mass, then starting slowly to move toward convergence. At beam 10000, NCP is still far from the quality of other algorithms. NCP greatly deteriorates the accuracy, showing that finding a reasonable linear time approximation algorithm is not a trivial task.

2.8.2 Speed Comparison

The speed comparison is done in terms of algorithm run-time. Table 2.3 reports the speed measures for beam = 30, comparing Standard Cube Pruning with the algorithms we propose. In the second column we report the average time spent
Table 2.3: Evaluation and comparison of the speed performance with beam size = 30. For each of the algorithms are reported: the average time for each sentence, the speed gain with respect to SCP, the average time spent on the Cube Pruning problem, the speed gain on CP with respect to SCP.

<table>
<thead>
<tr>
<th></th>
<th>avg. sent. t.</th>
<th>total t. var.</th>
<th>CP t.</th>
<th>CP t. var.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCP</td>
<td>312.8 ms</td>
<td>–</td>
<td>226.98 ms</td>
<td>–</td>
</tr>
<tr>
<td>FCP1</td>
<td>304.6 ms</td>
<td>5.91 %</td>
<td>202.90 ms</td>
<td>11.86 %</td>
</tr>
<tr>
<td>FCP2</td>
<td>305.2 ms</td>
<td>4.86 %</td>
<td>210.11 ms</td>
<td>8.02 %</td>
</tr>
<tr>
<td>FCP3</td>
<td>304.7 ms</td>
<td>5.49 %</td>
<td>205.97 ms</td>
<td>10.19 %</td>
</tr>
<tr>
<td>NCP</td>
<td>291.6 ms</td>
<td>10.26 %</td>
<td>194.02 ms</td>
<td>16.98 %</td>
</tr>
<tr>
<td>LCP</td>
<td>299.2 ms</td>
<td>7.45 %</td>
<td>197.41 ms</td>
<td>14.97 %</td>
</tr>
</tbody>
</table>

To translate a sentence. And in the third column we report the average time variation with respect to SCP.

As described in Dyer et al. (2010), HMT systems can be decomposed into a variety of subproblems and can be designed as a pipeline; the Cube Pruning is just a single module of the sequence of operations that compose the Machine Translation decoding. To have a precise idea of the speedup obtained using the proposed algorithms, we measured the average running time for the single module that executes the CP algorithms. We report this measure in the fourth column of Table 2.3. And in the fifth column we report the variation with respect to SCP.

SCP is the algorithm exploring the biggest portion of the search space, and is the slowest one. By comparing the overall runtime with the CP runtime, we notice that more than 2/3 of the time of the MT decoder is spent running the CP module. This shows that solving the CP problem is time consuming for a HMT model, and a significant reduction in runtime for the CP algorithm will result in an overall speedup for the MT decoder. FCP1 is significantly faster than SCP. FCP1 speeds up the CP module by 11.86%, and speeds up the overall translation by 5.58%. FCP2/3 are faster than SCP too. FCP2 speeds up the CP module by 8.02%, while FCP3 speeds up by 10.19%. From these results we can conclude that FCP3 is faster than FCP2 even if it explores the same portion of search space. This is due to the fact that FCP3 uses a simple boolean vector instead of a hash-map to check candidate collisions, as explained in section 2.6.3. On the other hand, FCP2/3 are not faster than FCP1, even if FCP1 explores a portion of the search space significantly bigger. This is due to the fact that FCP1 does not need any extra data-structure to check candidate collisions, as explained in
section 2.6.1. The NCP reaches a 16.98% speedup. This can be considered as a
higher bound on the speedup that can be reached with a linear time algorithm,
since NCP does not use additional data structures and is the simplest linear time
algorithm solving the CP problem. LCP speedup is 14.97%, significantly better
than the FCP algorithm and close to the NCP performance.

<table>
<thead>
<tr>
<th></th>
<th>avg. sent. t.</th>
<th>total t. var.</th>
<th>CP t.</th>
<th>CP t. var</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCP</td>
<td>814.4 ms</td>
<td>–</td>
<td>713.8 ms</td>
<td>–</td>
</tr>
<tr>
<td>FCP1</td>
<td>718.1 ms</td>
<td>13.40 %</td>
<td>624.2 ms</td>
<td>14.34 %</td>
</tr>
<tr>
<td>FCP2</td>
<td>757.3 ms</td>
<td>7.54 %</td>
<td>652.7 ms</td>
<td>9.34 %</td>
</tr>
<tr>
<td>FCP3</td>
<td>738.9 ms</td>
<td>10.22 %</td>
<td>636.3 ms</td>
<td>12.17 %</td>
</tr>
<tr>
<td>NCP</td>
<td>665.3 ms</td>
<td>22.40 %</td>
<td>570.9 ms</td>
<td>25.02 %</td>
</tr>
<tr>
<td>LCP</td>
<td>686.0 ms</td>
<td>18.71 %</td>
<td>586.2 ms</td>
<td>21.75 %</td>
</tr>
</tbody>
</table>

Table 2.4: Evaluation and comparison of the speed performance with beam size = 300. For each of the algorithms are reported: the average time for each sentence, the speed gain with respect to SCP, the average time spent on the Cube Pruning problem, the speed gain on CP with respect to SCP.

We repeat the same series of experiments with beam 300, and report the results in Table 2.4. By comparing the overall runtime with the time spent for the CP module, we notice that the portion of the time spent on the CP module grows with respect to the measure taken with beam 30. And it is easy to generalize this result and state that with higher beam a bigger portion of the time is spent on the CP module, and the overall execution time increases. We also notice that the relative variations increase, while the ranking of the algorithms is invariant.

To further investigate the speed comparison we plot the run time variation for all the algorithms for beam sizes in the range $[1, 10000]$ in Figure 2.11. We can confirm that the trends observed for beams 30 and 300 are consistent. All the presented algorithm are faster than SCP, and the relative speedup grows with the beam size. FCP1 is the faster among the FCP algorithms even if it explores a bigger portion of the search space. The linear time algorithms are considerably faster. While Figure 2.11 compares the speed variations with respect to SCP, Figure 2.12 plots relative speed gain with respect to LCP. This allow us to clearly show the asymptotic speed advantage of the linear time algorithms (LCP and NCP) over the $O(k \log k)$ algorithms (SCP and FCPs). Given the log-scale used to plot the beam size on the $x$-axis, the linear shape of the speed gain over FCP and SCP empirically confirms that LCP has a $log(k)$ asymptotic advantage.
2.8. Experimental Comparison on Approximately Monotonic CP

While, in contrast, the NCP speed gain converge to a horizontal line.

2.8.3 Speed/Accuracy Balance

So far we have discussed the speed and accuracy performances separately. Now we wish to investigate what speed/accuracy balance can be reached by the different algorithms. In Figure 2.13 we plot for each algorithm the speed/accuracy balance reached at different beam sizes: $[10, 20, 50, 100, 200]$. On the horizontal axis we plot the average time spent for each sentence on the CP module. On the vertical axis we plot the average search score of the top translation. We can see that the NCP (purple curve) performs significantly worse than the other algorithms after beam 10. We can also notice that LCP and the FCPs algorithms perform better than SCP (blue curve). The best speed/accuracy balances are reached by LCP and FCP1. LCP (orange curve) reaches the best speed/accuracy balance for beam lower than 100. Between k=100 and k=200, FCP1 surpasses LCP. This happens despite the fact that LCP is still faster than FCP1, because FCP compensate the speed loss with a better accuracy. FCP1 remains the algorithm
Figure 2.12: Run time variation relative to LCP for all the CP algorithms, at a different beam sizes in the range \([1, 10000]\), plotted on a log-scale.

with the best speed/accuracy balance for all the bigger beam sizes \([200, 10000]\). For bigger beam sizes the accuracy of all the algorithms tends to converge to the optimum. Once the algorithms reach a levels of accuracy non significantly different from the optimum, the speed performance is no longer relevant (e.g. all the curves became horizontal lines). In conclusion we can state that LCP and FCP1 can reach speed/accuracy balances that cannot be reached by the other CP algorithms. And this line of work does present algorithms that improve over the widely used SCP.

2.9 Discussion and Future Work

The proofs produced to show that the algorithms output the exact \(k\)-best list in the monotonic search space case do not apply to the approximately monotonic case. The empirical results reported show that the algorithms produce different outputs given the same input. Despite this difference, the accuracies reached are very close to each other.

To understand how the \(k\)-best lists produced by the algorithms differ at each
2.9. Discussion and Future Work

node of the hypergraph at decoding time, consider that in the approximately monotonic search space, it can happen that some element \( v \) is inserted into the \( k\)-best list, while some elements that are better than \( v \) are not yet in \( k\)-best. These events are due to the perturbations of the search space introduced by \( \Delta(\cdot) \), as explained in section 2.4.7. These cases are the cause of the divergence of behavior between the three algorithms. Since non-monotonicity is generally small, those elements that differ in the returned \( k\)-best lists will tend to be at the bottom of their respective \( k\)-best lists. Notice that the areas of non-monotonicity close to vertex, \( 1 \), of the search space will be entirely included in the \( k\)-best, and for these elements it does not matter in which order they have been added to the \( k\)-best, since the \( k\)-best will be sorted before passing it to the next level of the hypergraph. Thus we can assume that the top of the \( k\)-best lists contains only a small portion of the perturbations. And since the top part of the \( k\)-best will mainly influence the area near the vertex of the next level of the hypergraph, we can assume that most of the errors in the bottom of the \( k\)-best have a small probability to affect the output of the next node. This mechanism leads to discard most of the

Figure 2.13: Speed/Accuracy balance reached by the CP algorithms at different values of the beam size.
perturbations produced by the different algorithms, and allows to get as a result a high degree of similarity in the accuracy of the translations generated, as shown by the experimental results.

Future Work

This line of work can be extended in many directions. Useful insights can derive from the connection with related problems.

In Section 2.4.5 we formalized the connection between CP and the K-way merge problem. Following this intuition it is possible to think about finding new applications for the set of the presented algorithms as solution to classic Algorithmic problems such as K-way merge. And vice versa, the existing literature on K-way merge can provide ideas to be ported in the CP domain.

A similar reasoning can be done for any other problem that can be mapped to the CP problem. As we discussed CP pruning has already been connected to many existing problems such as alignment and $A^*$ search, resulting in finding new solutions to existing problems (such is the case of alignment) or extending the CP framework (such is the case of $A^*$ search). As an example, one could test the presented algorithm for any application of the CKY algorithm, such as Syntactic parsing.
In this chapter we present a novel Structured Prediction approach to Hierarchical Machine Translation (HMT). The novelty of the presented model is that it is not constrained to the standard bottom-up inference order. Removing the ordering constraint makes it possible to condition on top-down structure and surrounding context. This allows the introduction of a new class of contextual features that are not constrained to condition only on the bottom-up context. The model builds translation-derivations efficiently in a greedy fashion. It is trained to learn to choose jointly the best action and the best inference order. Experiments show that the decoding time is halved and forest-rescoring is 6 times faster, while reaching accuracy not significantly different from state of the art.

We refer to this new decoding approach to HMT decoding as Deterministic Machine Translation (DMT). The proposed model is deterministic in the sense that the decoder builds the structures incrementally, chooses a single action at each step, never retracts that action, and prunes all incompatible alternatives to that action. Deterministic decoding is made feasible by allowing translation-derivations to be constructed in an undirectional manner, so that the model can learn to choose the best inference order instead of being forced to use the bottom-up ordering imposed by the CKY-like beam-decoding used in standard HMT. Having pruning decisions based on a bottom-up approximation of contextual features leads to search errors that affect the quality of reordering and lexical-
Chapter 3. Deterministic Hierarchical Machine Translation

choice (Gesmundo and Henderson 2011). In contrast, DMT can choose to follow non-bottom-up decoding orders, thereby allowing the use of more context when making difficult decisions. By allowing many different decoding orders for the same tree structure, DMT can choose the ordering where it can be the most confident in its pruning decisions.

This undirected approach allows us to integrate contextual features such as the Language Model (LM) in a more flexible way. It also allows us to introduce a new class of undirected features. In particular, we introduce the Context-Free Factor (CFF) features, which exactly and efficiently compute a bound on the context-free cost of a partial derivation’s missing branches. CFF features help estimating the future cost of partial derivations. The new class of undirected features is fundamental for the success of a greedy/deterministic approach to MT, because access to additional non-bottom-up context is sometimes crucial to have the necessary information to make deterministic decoding decisions. DMT infers translation by building synchronous tree-structures as standard HMT. Thus DMT retains the advantages of HMT such as allowing complex reordering. On the other hand it applies a greedy/deterministic decoding strategy whose complexity is lower than standard HMT.

The DMT model selects the sequence of actions that incrementally build the translation using a discriminative scoring function. We introduce two alternative frameworks to train the parameters of the scoring function. The first proposed training framework focusses on the local context in which each action is selected without explicitly focussing on the future cost of the missing parts of the structure. While the second proposed training framework focusses on learning a decoding policy that optimizes a global loss function computed on complete structures, rather than maximizing the score of each individual action.

Experiments show that this model reduces overall execution time by more than half and forest-rescoring time by almost 10 times while reaching accuracy performance close to the state of the art for HMT.

3.1 Chapter Outline

In section 3.2, we review some of the previous work that has been influential for the definition of DMT, and that introduce some of the ideas that we are
generalizing to Structured Prediction and porting to HMT.

In section 3.3, we present a formalization of HMT as a generic structured prediction task, introducing basic ideas and concepts with a notation consistent with the following sections.

In section 3.4, we propose the Deterministic Machine Translation (DMT) model. We give a formal description of the framework and decoding algorithm. We detail further the model by describing an example of translation performed with DMT.

In section 3.5, we propose the Guided Structured Prediction (GSP) training framework. GSP is the first training method we propose to learn the parameters of DMT’s discriminative scoring function. The GSP algorithm tries to build the translation following the decoding algorithm. If an action contradicting the reference translation is selected, then a parameter update is triggered.

In section 3.6, we propose the Discriminative Reinforcement Learning (DRL) framework, the second training framework for DMT. DRL learns a deterministic decoding policy with the aim of optimizing a global loss function. DRL applies Reinforcement Learning (RL) techniques to structured prediction. Thus, we begin by reviewing the basics of RL. Then, we define a generic framework to train discriminative models for any structured prediction task using Reinforcement Learning. And finally we describe a DRL algorithm that can be applied specifically for DMT.

In section 3.7, we compare the two proposed training frameworks, underlining differences and similarities.

In section 3.8, we describe all the features that we apply for our experiments. We discuss how to adapt contextual features to the undirected approach. And we introduce new features that are available only for an undirected model.

In section 3.9 we report the experiments comparing the proposed models with standard HMT. The experiments show that the DMT decoder is twice as fast as standard HMT, and ten times faster while rescoring translations with the language model, at the cost of a small loss in translation quality.
Chapter 3. Deterministic Hierarchical Machine Translation

3.2 Related Work

Part of the content of this chapter has been presented in previous publications. The Discriminative Reinforcement Learning framework and its application to a Machine Translation decoder that has no inference order constraints has been presented for the first time in (Gesmundo and Henderson 2014). (Gesmundo and Henderson 2011) is the first model we presented that lifts the inference order from a Tree-based MT decoder. My previous work on Guided Learning applied to learning the inference order for sequence labeling tasks such as Part of Speech tagging, Named Entity recognition and Noun Phrase chunking, had a strong influence on line of work described in this chapter (Gesmundo 2011, 2009a,b).

Different deterministic approaches have been previously applied to other structure prediction tasks. For example, Nivre et al. (2006) presents a linear time syntactic dependency parser, which is constrained in a left-to-right decoding order. This model offers a different accuracy/complexity balance than the quadratic time graph-based parser (Mcdonald et al. 2005), but has never been shown to be a viable approach for HMT.

Many approaches to train greedy/deterministic models have been applied with success to other structured prediction tasks that are considered less complex than Tree-based Machine Translation. Shen et al. (2007) present the Guided Learning (GL) framework for bidirectional sequence classification. GL successfully combines the tasks of learning the order of inference and training the local classifier in a single Perceptron-like algorithm, reaching state of the art accuracy with complexity lower than the exhaustive counterpart (Collins 2002).

Goldberg and Elhadad (2010) present a similar training approach for a Dependency Parser that builds the tree-structure by recursively creating the easiest arc in a non-directional manner. This model also integrates the tasks of learning the order of inference and training the parser in a single Perceptron-like discriminative function. By “non-directional” they refer to the removal of the constraint of scanning the sentence from left to right, which is typical of shift-reduce models. However, this algorithm still builds the tree structures in a bottom-up fashion. This model has a $O(n \log n)$ decoding complexity and accuracy performance close to the $O(n^2)$ graph-based parsers Mcdonald et al. (2005).

Similarities can be found between the proposed models and previous work that applies discriminative training to structured prediction: Collins and Roark (2004)
3.3. **HMT as Structured Prediction**

Many tasks in natural language processing (NLP) can be formulated as mappings from inputs \( x \in \mathcal{X} \) to outputs \( y \in \mathcal{Y} \). Machine Learning (ML) solutions require learning a prediction function, \( f \), that maps input elements into the correct output, \( f : \mathcal{X} \rightarrow \mathcal{Y} \). Basic ML algorithms like Perceptron or SVM can be directly applied to solve tasks with simple output domains such as \( \mathcal{Y} = \{-1, +1\} \) (binary classification), \( \mathcal{Y} = \{C_1, \ldots, C_{|\mathcal{Y}|}\} \) (multiclass classification), \( \mathcal{Y} = \mathbb{R} \) (univariate regression). NLP tasks often require to output complex structures such as sequences of labels, tree-structures or graphs. Specialized solutions have been proposed for many NLP complex tasks, e.g.: sequence labeling (Brill 1995; Punyakanok and Roth 2001; Sutton et al. 2004); parsing (Collins 2003; Charniak and Johnson 2005); summarization (Knight and Marcu 2002; Barzilay 2003).

Machine Translation (MT) can also be addressed as a structured prediction task (Brown et al. 1993; Yamada and Knight 2001; Koehn et al. 2003). MT's goal is to learn a mapping function, \( f \), from an input sentence, \( x \), into \( y = \ldots \)
Chapter 3. Deterministic Hierarchical Machine Translation

\((t, h)\), where \(t\) is the sentence translated into the target language, and \(h\) is the hidden correspondence structure (Liang et al. 2006). In Hierarchical MT (HMT) (Chiang (2005, 2007)) the hidden correspondence structure is the synchronous-tree structure composed by instantiations of synchronous rules from the input grammar, \(G\).

Statistical models usually define \(f\) as: 
\[
\begin{align*}
\arg \max_{y \in \mathcal{Y}} \text{score}(x, y),
\end{align*}
\]

where \(\text{score}(x, y)\) is a function whose parameters can be learned with a specialized learning algorithm. For instance, Collins (2002) solves the argmax exhaustively for sequence labeling, by applying Viterbi decoding. When exhaustive search is not tractable, beam search techniques can be applied. Such is the case for MT, in practice it is not possible to enumerate all \(y \in \mathcal{Y}\). For HMT, decoding applies substantial pruning such as Cube Pruning techniques (Huang and Chiang 2005), Gesmundo and Henderson (2010), Gesmundo et al. (2012). But even using pruning, HMT has higher complexity than Phrase-based MT (PbMT) (Koehn et al. (2003)). On the other hand HMT improves over PbMT by exploiting a more sophisticated reordering model not bounded by a window size, and producing translations with higher syntactic-semantic quality. In this chapter we propose Deterministic Machine Translation (DMT), which retains the advantages of HMT, such as allowing complex reordering, and allows the use of a deterministic decoder whose complexity is lower than standard HMT.

### 3.4 Deterministic Machine Translation

In this section, we present the DMT framework. For clarity of presentation and following Tree-based MT practice, we will henceforth restrict our focus to binary grammars (Zhang et al. (2006); Wang et al. (2007)).

A DMT decoder can be formulated as a function, \(f\), that maps a source sentence, \(x \in \mathcal{X}\) into a structure defined by \(y = (t, h) \in \mathcal{Y}\), where \(t\) is the translation in the target language, and \(h\) is the synchronous tree structure generating the input sentence on the source side and its translation on the target side. Synchronous-trees are composed by instantiations of synchronous rules, \(r\), from a grammar, \(G\).

A DMT decoder builds synchronous-trees, \(h\), by recursively expanding partial synchronous-trees, \(\tau\). \(\tau\) includes a partial translation. Each \(\tau\) is required to be
3.4. Deterministic Machine Translation

a connected sub-graph of some complete synchronous-tree \( h \) that generates \( x \) on the source side. Thus, \( \tau \) is composed of a subset of the rules from any \( h \), such that there is a connected path between any two rules in \( \tau \). There is no constraint regarding the set of terminals included in \( \tau \). Differently from the partial structures built by a bottom-up decoder, \( \tau \) does not have to cover a contiguous span on \( x \). Alternatively, we can think of \( \tau \) as being obtained by taking a subtree of some complete synchronous-tree \( t \) and then pruning some of its branches.

Formally, \( \tau \) is defined by:

1. The set of synchronous-rule instantiations in \( \tau \):

\[
I \equiv \{r_1, r_2, \ldots, r_k \mid r_i \in G, 1 \leq i \leq k\} \tag{3.1}
\]

2. The set of connections among the synchronous-rule instantiations, \( C \). Let \( c_i = (r_i, r_{ji}) \) be the notation to represent the connection between the i-th rule and the rule \( r_{ji} \). The set of connections can be expressed as:

\[
C \equiv \{(r_1, r_{j1}), (r_2, r_{j2}), \ldots, (r_{k-1}, r_{jk-1})\} \tag{3.2}
\]

Notice that \( |I| = |C| + 1 \).

3. The postcondition set, \( P \), which specifies the non-terminals in \( \tau \) that are available for creating new connections. Each postcondition, \( p_i = (r_x, X_{[y]}c_i) \), indicates that the rule \( r_x \) has the non-terminal \( X_{[y]} \) available for connections. The index \( c \) identifies the non-terminal in the rule. In a binary grammar \( \beta \) can take only 3 values: \( \beta \) for the first non-terminal (the left child of the source side), \( \gamma \) for the second non-terminal, (the right child of the source side), and \( \eta \) for the head. The postcondition set can be expressed as:

\[
P \equiv \{(r_{x1}, X_{[y1]}c_1), (r_{x2}, X_{[y2]}c_2), \ldots, (r_{xm}, X_{[ym]}c_m)\} \tag{3.3}
\]

Notice that if the grammar is binary and \( \tau \) has at least one leaf then: \( |P| \leq |I| \)

4. The set of carries, \( K \). We define a different carry, \( \kappa_i \), for each non-terminal available for connections. Each carry stores the extra information required to correctly score the non-local interactions between \( \tau \) and the rule that will
be connected at that non-terminal. Thus $|K| = |P|$. Let $\kappa_i$ be the carry associated with the postcondition $p_i$. The set of carries can be expressed as:

$$K \equiv \{\kappa_1, \kappa_2, \cdots, \kappa_m\}$$  \hspace{1cm} (3.4)

Partial synchronous-trees, $\tau$, are expanded by performing connection actions. Given a $\tau$ we can connect to it a new rule, $\hat{r}$, using one available non-terminal represented by postcondition, $p_i \in P$, and obtain a new partial synchronous-tree $\hat{\tau}$. We denote the connection action with the notation:

$$\hat{\tau} \equiv \langle \tau \leftarrow \hat{a} \rangle$$  \hspace{1cm} (3.5)

Where, $\hat{a} = [\hat{r}, p_i]$, represents the connection action. We denote the set of all partial synchronous-trees with $S$. We denote the set of all actions that can be applied to expand the partial synchronous-tree $\tau$ with $A_\tau$.

### 3.4.1 Decoding Algorithm

Having defined the partial synchronous-trees $\tau$, we proceed by describing the algorithm that is used by the DMT decoder to build a finite translation by iteratively expanding the partial structure using connection actions. We refer to this algorithm as the DMT decoding algorithm.

The pseudocode of the DMT decoding algorithm is listed in Algorithm 8. The decoder takes as input the source sentence, $x$, the synchronous-grammar, $G$, and a scoring function, $\text{score}(\cdot) : S \times A_\tau \rightarrow \mathbb{R}$, mapping structure-action pairs into scalar values. At line 2 the partial synchronous-tree $\tau$ is initialized by setting $I$, $C$, $P$ and $K$ to empty sets $\emptyset$. At line 3 the queue of candidate connection actions is initialized as:

$$Q \equiv \{ [r_{leaf}, \text{null}] \mid r_{leaf} \text{ is a leaf rule instantiation} \}$$  \hspace{1cm} (3.6)

Where $\text{null}$ means that there is no postcondition specified, since the first rule does not need to connect to anything and at this stage $\tau$ has no postconditions available. A leaf rule $r_{leaf}$ is any synchronous rule with only terminals on the right-hand side. At line 4 the main loop starts. Each iteration of the main loop will expand $\tau$ using one connection action. The loop ends when $Q$ is empty,
Algorithm 8  DMT Decoding

1: function Decoder ($x$, score($\cdot$), $G$) : $(t,h)$
2: $\tau.\{I, C, P, K\} \leftarrow \{\emptyset, \emptyset, \emptyset, \emptyset\}$ ;
3: $Q \leftarrow \text{LeafRules}(G, x)$;
4: while $|Q| > 0$ do
5: $[\hat{r}, p_i] \leftarrow \text{PopBestAction} (Q|\text{score}(\cdot))$;
6: $\tau \leftarrow \text{CreateConnection}(\tau, [\hat{r}, p_i])$;
7: $\text{UpdateQueue}(Q, [\hat{r}, p_i])$;
8: end while
9: Return$(\tau)$;

10: procedure CreateConnection($\tau, [\hat{r}, p_i]$) : $\hat{\tau}$
11: $\hat{\tau}.I \leftarrow \tau.I + \hat{r}$;
12: $\hat{\tau}.C \leftarrow \tau.C + (\hat{r}, r_{p_i})$;
13: $\hat{\tau}.P \leftarrow \tau.P - p_i$;
14: $\hat{\tau}.K \leftarrow \tau.K - \kappa_i$;
15: $\hat{\tau}.K.\text{UpdateCarries}(\hat{r}, p_i)$;
16: $\hat{\tau}.P.\text{AddAvailableConnectionsFrom}(\hat{r}, p_i)$;
17: $\hat{\tau}.K.\text{AddCarriesForNewConnections}(\hat{r}, p_i)$;
18: Return$(\hat{\tau})$;

19: procedure UpdateQueue( $Q, [\hat{r}, p_i]$ ) :
20: $Q.\text{RemoveActionsWith}(p_i)$;
21: $Q.\text{AddNewActions}(\hat{r}, p_i)$;

implying that $\tau$ covers the full sentence and has no more missing branches. The best scoring action according to the scoring function is popped from the queue at line 5. At line 6 the selected connection action is used to expand $\tau$. At line 7 the queue of candidates is updated accordingly (see lines 19-21). At line 8 the decoder iterates the main loop, until $\tau$ is complete and is returned at line 9 as $(t,h)$.

Lines 10-18 describe the CreateConnection($\cdot$) procedure, that connects the partial synchronous-tree $\tau$ to the selected rule $\hat{r}$ via the postcondition $p_i$ specified by the candidate-action selected in line 5. This procedure returns the resulting partial synchronous-tree: $\hat{\tau} \equiv \{ \tau \leftarrow [\hat{r}, p_i]\}$. At line 11, $\hat{r}$ is added to the rule set $I$. At line 12, the connection between $\hat{r}$ and $r_{p_i}$ (the rule specified in the postcondition) is added to the set of connections $C$. At line 13, $p_i$ is removed from $P$. At line 14, the carry $k_i$ matching with $p_i$ is removed from $K$. At line 15, the
set of carries $K$ is updated, in order to update those carries that need to provide information about the new action to correctly score rules that will be connected using the relative postcondition. As an example, for the LM information contained in the carry, if the newly connected rule introduces new target terminals, then those carries in $K$ whose state depends on this new information need to be updated, by adding previously unknown boundary words. At line 16, new postconditions representing the non-terminals in $\hat{r}$ that are available for subsequent connections are added in $P$. At line 17, the carries associated with these new postconditions are computed and added to $K$. Finally at line 18, the updated partial synchronous-tree is returned.

Notice that, in the very first iteration, the CreateConnection($\cdot$) procedure has nothing to compute for some of the lines: line 11 is not executed since the first leaf rule needs no connection and has nothing to connect to. lines 12-13 are not executed since $P$ and $K$ are $\emptyset$ and $p_i$ is not specified for the first action. line 15 is not executed since there are no carries to be updated. lines 16-17 only add the postcondition and carry relative to the leaf rule head link.

The procedure used to update $Q$ is reported in lines 19-21. At line 20, all the connection actions involving the expansion of $p_i$ are removed from $Q$. These actions are the incompatible alternatives to the selected action, and thus need to be removed. Notice that, during the very first iteration, all actions in $Q$ are removed because they are all incompatible with the connected-graph constraint. At line 21, new connection actions are added to $Q$. These are the candidate actions proposing a connection to the available non-terminals of the selected action’s rule, $\hat{r}$. The rule instantiations used for these new candidate-actions must not be in conflict with the current structure of $\tau$ (e.g. the rule cannot generate a source side terminal that is already covered by $\tau$).

Notice that at line 3, we initialize $Q$ using only leaf rules. This leads the decoder to start building the structure starting from a leaf. But in general, the decoder is not required to have this constraint. We constrain the model to start from a leaf because it makes sense for tasks like HMT, that have a small number of nonterminal symbols. And in addition, this constraint lowers the complexity of the initialization. But for other applications this constraint might not be justifiable and the constraint can be lifted.
3.4. Deterministic Machine Translation

\[
\begin{align*}
  r_a : X &\to X_0 X_1 | X_0 X_2 \\
  r_b : X &\to we | nous \\
  r_c : X &\to go | vais \\
  r_d : X &\to go | vas \\
  r_e : X &\to go | allons \\
  r_f : X &\to go | allez \\
  r_g : X &\to go | vont \\
\end{align*}
\]

Figure 3.1: Example of binary synchronous-grammar for HMT.

3.4.2 DMT Decoding Example

So far we have described the Deterministic Machine Translation framework and decoding algorithm in abstract terms. Now we wish to explain it in more details associating it to a practical application. In the following example, we show how the DMT framework can be used to produce synchronous-tree structures encoding a translation. We describe in details the execution of the algorithm on the translation of a simple sentence.

Figure 3.2 depicts the sequences of states of a DMT decoder translating a sentence from English to French. The decoder input is the synchronous-grammar listed in Figure 3.1 and an input sentence containing the words “we go”:

\[
x = < w_1, \cdots, “we”, “go”, \cdots, w_n >
\]  
(3.7)

After the initialization steps, Q contains a candidate starting-action for each leaf-rule matching the terminals in \( x \). Examples of leaf-rules are \( r_b, r_c, r_d, r_e, r_f \) and \( r_g \).

The decoder chooses to perform the action it is most confident in. Assume that at the first iteration, the decoder decides to translate “we” into “nous”. Translating “we” is less ambiguous than translating “go”, since “go” must be conjugated in agreement with the subject. The \( \tau \) resulting after the first iteration is depicted in Figure 3.2a: \( \tau.I \) contains only \( r_1 \), an instantiation of rule \( r_b \in G \) (note that we use letter indices for rules in \( G \) and number indices for rule-instantiations); \( \tau.C \) is still empty, since no connections have been established; \( \tau.P \) contains only the postcondition, \( p_1 \equiv (r_1, X_0) \), representing the available head-connection of \( r_1 \); \( \tau.K \) contains the carry, \( \kappa_1 \), associated with \( p_1 \). The carry
stores the extra information required to correctly score the non-local interactions between the current partial subtree and the rule that will be connected at the non-terminal represented by $p_1$. For this example, the only contextual feature considered is a 3-gram LM. Thus $\kappa_1$ contains the boundary words information to be passed bottom-up through the head-connection of $r_1$. For the subtree composed by $r_1$, the only information needed is the translation of $\tau$’s yield into the word “nous”. This information is represented with the notation $\kappa_1 = ("nous")$. This notation follows the standard used for bottom-up decoders (Chiang (2007)): for cases in which an $X$-gram LM model is used and the current substructure covers a contiguous span on the target side of length smaller than $X$, then all the target words are stored in the carry. In our case, we use a 3-gram model, and the current substructure covers a single word on the target side. Notice that the action performed at the first iteration is equivalent to a bottom-up action since the decoder starts from a leaf.
3.4. Deterministic Machine Translation

At the beginning of the second iteration, \( Q \) contains all the possible ways to expand the head non-terminal of \( r_1 \). Figure 3.2b depicts the state of \( \tau \) at the end of the second iteration: \([r_2, p_1]\) has been chosen as best action; \( r_2 \) has been added in \( \tau.I\); \( \tau.C \) contains the connection between \( r_1 \) and \( r_2 \); \( \tau.P \) contains the two new postconditions \( p_1 \) (representing the available head-connection of \( r_2 \)) and \( p_2 \) (representing the available right-child-connection of \( r_2 \)); and \( \tau.K \) contains their respective carries \( \kappa_1 \) and \( \kappa_2 \).

The notation \( \kappa_1 \equiv ([nous \ ? \ ? \ ?])_1 \) is an extension of the standard star notation used to represent the LM information contained in a HMT bottom-up carry (Chiang (2007)). We introduce the symbol “?” as placeholder for words that are currently unknown. “[nous \ ? \ ? \ ?]” means that the first word after the left boundary is “nous”, while the second word is unknown. Since we are using a 3-gram LM, only the first two boundary words are relevant to score interaction with surrounding context. The symbol * is used as placeholder for irrelevant words between the boundary words. The two words close to the right boundary are also unknown. With the second carry, \( \kappa_2 \equiv (\ ? \ nous [-] \ ? \ ?)_2 \), we introduce a new notation for top-down connections. The top-down LM information concerns the boundary words surrounding the child’s span. “? nous [-] ? ?” means that the word at the left of the child’s span is “nous”, and all the other surrounding words are unknown.

At the beginning of the third iteration, \( Q \) contains all the possible ways to expand \( r_2 \)’s head and right-child non-terminals. At this step it is easy to choose the correct translation for “go” using the information carried by \( \kappa_2 \). Since “nous allons” is a frequent bigram in French, the action translating “go” into “allons” has a high score. Figure 3.2c shows the state of \( \tau \) after choosing this action as the third action. Notice that the carry related to the head connection of \( r_2 \) has been updated accordingly to \( \kappa_1 \equiv (nous \ allons) \), storing the two boundary words needed to score the LM interaction when a connection is established via the postcondition \( p_1 \).

3.4.3 Discriminative Scoring Function

So far we have defined DMT using a generic scoring function:

\[
\text{score} (\tau) : S \times A_\tau \rightarrow \mathbb{R}
\]

(3.8)
Since we will focus on discriminative training methods, we now specify the scoring function as a discriminative function:

$$\text{score}(\tau) = \mathbf{w} \cdot \phi(\tau, a) \quad \forall \tau \in S, a \in A_{\tau}$$

(3.9)

Where: \( \mathbf{w} \) is the weight vector and \( \phi(\cdot) \) is the feature function. The feature function is used to map state-action pairs, \((\tau, a)\), to a real valued vector.

$$\phi(\cdot) : S \times A_{\tau} \rightarrow \mathbb{R}^d$$

(3.10)

Where \( d \) is the number of features applied. The weight vector is required to have the same number of dimensions as \( \phi(\tau, a) \): \( |\mathbf{w}| = d \). Thus, each entry of the weight vector is the weight of the matching feature. Each dimension corresponds to a distinct feature function that maps: \( S \times A_{\tau} \rightarrow \mathbb{R} \). Details of the features implemented for our model are given in section 3.8. This setting is typical of discriminative models for structured prediction (Collins 2002).

The techniques used to learn the weight vector can vary. The interpretation of the value returned by the scoring function changes accordingly to the training approach. In the following sections, we are going to propose two distinct training techniques to learn the parameters of the discriminative training function. In section 3.5, we present the Guided Structured Prediction training framework, that focusses on optimizing each individual action selection given the local context. While in section 3.6, we describe the Deterministic Reinforcement Learning training framework, that focusses on learning a decoding policy that optimizes a global loss function. In section 3.7, we discuss and compare the two proposed frameworks.

### 3.5 Guided Structured Prediction

In this section we propose the Guided Structured Prediction (GSP) training framework. GSP is the first training method we propose for DMT. The specificity of the GSP training framework lies in its ability to learn the order of inference together with the parameters of the discriminative scoring function instead of forcing the decoder into a pre-defined order. Given a sentence in the source language and the matching reference translation, the GSP algorithm attempts to
build the candidate synchronous-tree following the DMT decoding algorithm. At each iteration a new action is selected according to the current scoring function. If the selected action leads to a partial structure that cannot generate the reference translation, then the weight vector is updated penalizing the features of the wrong action and promoting the features of the highest scoring action that would have not conflicted with the correct translation. The updates of the weight vector can be performed with any discriminative training technique such as the Perceptron Algorithm.

GSP can be regarded as an application of the Guided Learning framework (Shen et al. 2007) to the DMT framework. While Guided Learning has been proposed for sequence tagging, GSP can be applied to more complex structures such as synchronous-trees. Furthermore, Guided Learning was proposed with a specific Aggressive Update strategy. Instead, we define GSP without any constraint on the update strategy. This generic approach allows us to plug-in and test different discriminative update techniques, as we will discuss in section 3.7.

Algorithm 9 summarizes the GSP learning algorithm. The trainer takes as input:

1. The feature-vector function $\phi(\cdot)$, mapping a pair $(\tau, a)$ to a real valued vector.

2. The weighted synchronous grammar, $G$.

3. The data $D$, that consists of a set of pairs of sentences, $D \equiv \{(x, t^*)\}$, where $x$ is the source sentence and $t^*$ is the reference translation.

The algorithm returns the weight vector, $w$, that specifies the parameters of the discriminative scoring function to be used at decoding time.

The external loop, lines 2-18, runs as many training epochs as are needed to reach convergence. The loop between lines 3-17 runs a single training epoch over all the data set $D$, by taking in consideration a single sample $(x, t^*) \in D$ at each iteration.

At lines 4-5, $\tau$ and $Q$ are initialized as in the DMT decoding algorithm (Algorithm 8). The partial synchronous-tree $\tau$ is initialized by setting $I$, $C$, $P$ and $K$ to empty sets $\emptyset$. And the queue of candidate connection actions is
Algorithm 9  GSP Training

1: function Trainer (φ, G, D) : w
2: repeat
3:   for (x, t*) ∈ D do
4:     τ. {I, C, P, K} ← {∅, ∅, ∅, ∅};
5:     Q ← LeafRules(G, x);
6:   while |Q| > 0 do
7:     a ← GetBestAction(Q|φ);
8:     if a.DoesNotMatch(t*) then
9:       a* ← GetBestCorrectAction(Q, t*);
10:      w. Update(-a, +a*);
11:     RescoreQueue(Q);
12:   end if
13:   [ˆr, p_i] ← PopBestAction(Q);
14:   τ ← CreateConnection(τ, [ˆr, p_i]);
15:   UpdateQueue(Q, [ˆr, p_i]);
16: end while
17: until convergence
18: Return(w);

20: procedure CreateConnection(τ, [ˆr, p_i]) : ˆτ
21:   ˆτ.I ← τ.I + ˆr;
22:   ˆτ.C ← τ.C + (ˆr, r_p_i);
23:   ˆτ.P ← τ.P − p_i;
25:   ˆτ.K.UpdateCarries(ˆr, p_i);
26:   ˆτ.P.AddAvailableConnectionsFrom(ˆr, p_i);
27:   ˆτ.K.AddCarriesForNewConnections(ˆr, p_i);
28: Return(ˆτ);

29: procedure UpdateQueue( Q, [ˆr, p_i]) :
30:   Q.RemoveActionsWith(p_i);
31:   Q.AddNewActions(ˆr, p_i);
initialized as:

\[ Q \equiv \{ [r_{leaf}, \textbf{null}] \mid r_{leaf} \text{ is a leaf rule instantiation} \} \] (3.11)

Where \textbf{null} means that there is no postcondition specified, since the first rule does not need to connect to anything and at this stage \( \tau \) has no postconditions available. A leaf rule \( r_{leaf} \) is any synchronous rule with only terminals on the right-hand sides.

At line 6, the inner training loop starts. The loop is interrupted if \( Q \) is empty. At line 7, a pointer to the current top scoring candidate-action is stored in \( a \), but the element pointed to by \( a \) is not removed from the queue. At line 8, the algorithm checks if \( a \) is a correct choice, by checking if it contradicts the target translation \( t^* \). If \( a \) is not correct, the weight vector is updated executing lines 9-11. At line 9, a pointer to the top scoring correct action is stored in \( a^* \). With “correct actions” we refer to those actions that will generate a partial structure that is still capable of generating the target translation. At line 10, the perceptron weights are updated, promoting the features of \( a^* \) and penalizing the features of \( a \). There can be some situations where there are no correct actions in \( Q \), in which case the update is not performed. Notice that in this generic definition of the GSP Training algorithm we leave the details of the update unspecified. This allows one to plug-in at line 10 the desired discriminative update technique, such as any specific perceptron update technique. At line 11 the queue is rescored using to the updated scoring function. At line 13 the best action is popped from \( Q \). At line 14, \( \tau \) is expanded with the new action using the procedure CreateConnection(·) described for the DMT decoder (Algorithm 8). And finally at line 15, \( Q \) is updated using the procedure UpdateQueue(·) described for the DMT decoder (Algorithm 8). After reaching convergence, the weight vector \( w \) is returned at line 19.

Notice that, if the parameter update is not performed, the action popped at line 13 is \( a \). Otherwise the action popped is whatever action is preferred by the updated scoring function. The generic GSP algorithm does not guarantee that the action popped at line 13 is a correct action. This constraint can be enforced by plugging at line 10 any update function that guarantee that \( a^* \) will score better than all other actions (e.g. Passive-Aggressive: Crammer et al. (2006)). Regardless, training in a setting in which the action popped may be wrong makes the training closer to the decoding setting, and allows the system to learn to
choose reasonable actions after a mistake has been made. We will compare and discuss the different alternatives for the update function in section 3.7.1.

In this section we presented GSP as the first training technique for DMT. GSP trains the parameter of the discriminative function focussing on the local context in which each action is selected without explicitly focussing on the future cost of the missing parts of the structure. This approach may lead DMT to choose an action with a good local score but that will prevent it from reaching the structure with highest global score. In the next section we present an alternative training technique for DMT, that focuses more on maximizing the global score of the complete structure rather than maximizing the score of each individual action.

3.6 Discriminative Reinforcement Learning

In this section we propose the Discriminative Reinforcement Learning (DRL) framework. DRL is the second training method we propose for DMT. DRL updates the weight vector with the goal of maximizing the global score of the complete structure rather than maximizing the score of each individual action. Because each action choice must be evaluated in the context of the complete translation-derivation, we formalize this method in terms of Reinforcement Learning (RL). We begin by reviewing the basics of RL. Then, we define a generic framework to train discriminative models for any structured prediction task using Reinforcement Learning. And finally we detail a DRL algorithm that can be applied specifically for DMT.

3.6.1 Reinforcement Learning Review

In general, Reinforcement Learning can be applied to any task, $\mathcal{T}$, that can be formalized in terms of $\mathcal{T} \equiv \{S, \{A_s\}, T, R, \gamma\}$, where:

1. $S$ is the set of states. $S$ can be either finite or infinite.

2. $A_s$ is a set of actions for each state $s \in S$.

3. $T : S \times A_s \to S$, is the transition function, that specifies the next state given a source state and performed action$^1$.

$^1$For simplicity we describe a deterministic process. To generalize to the stochastic process, replace the transition function with the transition probability: $P_{sa}(s')$, $s' \in S$. 

Chapter 3. Deterministic Hierarchical Machine Translation
4. \( R : S \times A_s \rightarrow \mathbb{R} \), is the reward function.

5. \( \gamma \in [0, 1] \) is the discount factor.

The solution of the task can be expressed as a mapping from states to actions. The mapping is formally referred to as a policy, \( \pi : S \rightarrow A \). Given a RL task there could be any number of policies that can be proposed as a solution. A RL agent is required to learn the policy that maximizes the sum of rewards received when starting from some initial state and proceeding to a terminal state.

Let us denote with: \( \text{path}(s_0|\pi) \equiv \langle s_0, s_1, \ldots, s_\sigma | \pi \rangle \), the sequence of states obtained by following the policy \( \pi \), starting at the initial state \( s_0 \), up to a terminal state \( s_\sigma \).

The value function for a policy \( \pi \) evaluated at state \( s_0 \) is defined by:

\[
V^\pi(s_0) = \sum_{i=0}^{\sigma} \gamma^i R(s_i, \pi(s_i)) \tag{3.12}
\]

Meaning that, given a policy, \( \pi \), we can associate a value, \( V^\pi(\cdot) \), to any state, \( s_0 \), where the value \( V^\pi(s_0) \) is equal to the sum of the rewards received starting at that state and following \( \pi \) up to a terminal state. The discount factor \( \gamma \) is used to exponentially decrease the weight of rewards received in the future. \( \gamma \) has values in the range \([0, 1]\). If \( \gamma = 1 \) there is no discount. The lower the value of \( \gamma \) the higher importance immediate rewards have than future rewards. The optimal policy would therefore be the mapping from states to actions that maximizes the sum of the rewards when starting in an arbitrary state and performing actions until a terminal state is reached. Under this definition the value of a state is dependent upon the policy. The value function can be interpreted as a mapping from states to state values.

To distinguish between the different actions available at a certain state, and possibly score and rank these actions, the \( Q \)-function is defined:

\[
Q^\pi(s, a) = R(s, a) + \gamma V^\pi(T(s, a)) \tag{3.13}
\]

Given a start state, \( s \), and a policy, \( \pi \), the \( Q \)-function associates a \( Q \)-value to each possible action that can be performed from \( s \). The \( Q \)-value is the reward obtained by performing the action \( a \in A_s \), summed to \( \gamma V^\pi(T(s, a)) \), where \( T(s, a) \) stands for the state reached after performing \( a \) from state \( s \).
The optimal policy, $\pi^*$, is the one that maps each state, $s$, to the optimal action $a^*$, maximizing the future reward:

$$\pi^*(s) = a^* = \arg\max_{a \in A_s} \; Q^*(s, a) \quad \forall s \in S$$  \hspace{1cm} (3.14)

The $Q$-function is a mapping from (state, action) pairs to scalar values. The $Q$-function can be approximated using any type of function approximator. We will discuss in more details the use of discriminative function approximators.

### 3.6.2 Discriminative Reinforcement Learning Generic Framework

Because we are taking a discriminative approach to learn the function approximator, we formalize our optimization task similarly to an inverse reinforcement learning problem (Ng and Russell 2000): we are given information about the optimal global loss function and we want to learn a discriminative reward function. As in other discriminative approaches, this approach simplifies the task of learning the reward function in two respects: the learned reward function only needs to be monotonically related to the true reward function, and this property only needs to hold for the best competing alternatives. This is all we need in order to produce a good approximation of the optimal policy using a discriminative approximation of the $Q$-function. This simplification makes learning easier in cases where the true reward function is too complicated to model directly.

We assume that the optimal $Q$-function can be approximated with a linear discriminative function $\tilde{Q}(s,a)$ in some feature-vector representation $\phi : S \times A_s \rightarrow \mathbb{R}^d$ that maps a state-action pair to a $d$-dimensional features vector:

$$\tilde{Q}(s,a) = w \cdot \phi(s,a)$$  \hspace{1cm} (3.15)

where $w \in \mathbb{R}^d$ is the weight vector. This gives us the following approximation of the optimal policy:

$$\pi_w(s) = \arg\max_{a \in A_s} \; w \cdot \phi(s,a)$$  \hspace{1cm} (3.16)

The set of parameters of this policy is the vector $w$. With this formalization, all we need to learn is a vector $w$ such that the resulting decisions are compatible with the given information about the optimal sequence of actions. We propose a
Algorithm 10 Discriminative Reinforcement Learning

1: function Trainer (φ, T, D) : w
2: repeat
3:    s ← SampleState(S);
4:    ̂a ← πw(s);
5:    a′ ← SampleAction(A);
6:    if Qπw(s, ̂a) < Qπw(s, a′) given D then
7:        w ← w + Φw(s, a′) − Φw(s, ̂a);
8:    end if
9: until convergence
10: Return(w);

Perceptron-like algorithm to learn these parameters.

Algorithm 10 describes the Discriminative Reinforcement Learning (DRL) meta-algorithm. The training algorithm takes as input the feature function φ, the task T, and a generic set of data D describing the behaviors we want to learn. The output is the weight vector w defining the learned policy that fits the data D. The algorithm consists of a single training loop that is repeated until convergence (lines 2-9). At line 3, a state, s, is sampled from S. At line 4, ̂a is set to be the action that would be preferred by the current policy πw. At line 5, an action, a′, is sampled from A such that a′ ̸= ̂a. At line 6, the algorithm checks if preferring path(T(s, ̂a), πw) over path(T(s, a′), πw) is a correct choice according to the behaviors data D that the algorithm aims to learn. If the current policy πw contradicts D, line 7 is executed to update the weight vector to promote Φw(s, a′) and penalize Φw(s, ̂a). Where Φw(s, a) is the summation of the features vectors of the entire derivation path starting at (s, a) and following policy πw.

\[
\Phi^w(s, a) = \phi(s, a) + \sum_{s_i \in \text{path}(T(s, a), \pi_w)} \phi(s_i, \pi_w(s_i)) \quad (3.17)
\]

This way of updating w has the effect of increasing the \( \tilde{Q}(\cdot) \) value associated with all the actions in the sequence that generated the promoted structure, and reducing the \( \tilde{Q}(\cdot) \) value of the actions in the sequence that generated the penalized structure\(^2\). The goal of this update approach is to make \( \tilde{Q}(\cdot) \) closer to the real Q(·) function determined by the data D.

\(^2\)Preliminary experiments with updating only the features for ̂a and a′ produced substantially worse results.
We have described the DRL meta-algorithm to be as general as possible. When applied to a specific problem, more details can be specified:

- It is possible to choose specific sampling techniques to implement lines 3 and 5;
- The test at line 6 needs to be detailed according to the nature of $\mathcal{T}$ and $D$;
- The update statement at line 7 can be replaced with a more sophisticated update approach.

We address these issues and describe a range of alternatives as we apply DRL to DMT in the next section.

### 3.6.3 Application of Discriminative Reinforcement Learning to Deterministic Machine Translation

To apply the Discriminative Reinforcement Learning framework to learn the parameters of a Deterministic Machine Translation model, we formalize DMT as a DRL task $\mathcal{T} \equiv \{S, \{A_s\}, T, R, \gamma\}$:

1. The set of states $S$ is the space of all possible DMT partial synchronous-trees, $\tau$, matching the input sentence $x$ on the source side.
2. The set $A_\tau$ is the set of connection actions that can expand $\tau$ connecting new synchronous-rule instantiations.
3. The transition function, $T : S \times A_s \rightarrow S$, is the connection function $\hat{\tau} \equiv \langle \tau \leftarrow a \rangle$ formalized in section 3.4 and detailed by the procedure CreateConnection($\cdot$) in the DMT decoding algorithm (Algorithm 8).
4. As reward function, $R : S \times A_s \rightarrow \mathbb{R}$, we choose to use the BLEU score. We will evaluate and compare the accuracy of the model using the BLEU score. Therefore we set this metric to be the reward function to be learnt. BLEU is a function that quantifies the difference between the reference translation $t^*$ and the output translation $t$ (Papineni et al. 2002).

The BLEU score can be computed only when a terminal state is reached and a full translation is available. Thus, the rewards are all zero except
Algorithm 11 Discriminative Reinforcement Learning for DMT

1: function Trainer ($\phi, T, D$) : $w$
2: repeat
3: for $(x, t^*) \in D$ do
4:   $\tau \leftarrow \text{UniformSample}(\text{path}(s_0, \pi|x))$;
5:   $\hat{a} \leftarrow \arg \max_{a \in A_\tau} \hat{Q}_{\pi w}(\tau, a)$;
6:   $a' \leftarrow \text{ZipfianSample}(\{\{A_\tau - \hat{a}\} \text{ sorted with } \hat{Q}_{\pi w}(\cdot)\})$;
7:   if $\text{BLEU}(t^*, t(\tau, \hat{a})) < \text{BLEU}(t^*, t(\tau, a'))$ then
8:     $w \leftarrow w + \Phi^w(\tau, a') - \Phi^w(\tau, \hat{a})$;
9:   end if
10: end for
11: until convergence
12: Return($w$);

at terminal states. This type of reward functions are referred to as Pure Delayed Reward function.

5. Considering the nature of the problem and reward function, we choose an undiscounted setting: $\gamma = 1$.

Notice that, given a partial structure $\tau$, the set of actions $A_\tau$ includes only the actions such that: $\hat{\tau} \in S$, with $\hat{\tau} \equiv \langle \tau \leftarrow a \rangle$ and $a \in A_\tau$, meaning that a partial structure $\tau$ can be extended only with actions resulting in a new structure $\hat{\tau}$ that matches the input sentence $x$ on the source side. This constraint greatly reduces the size of the set $A_\tau$ and of the search space.

Algorithm 11 details the application of the Discriminative Reinforcement Learning meta-algorithm to the Deterministic Machine Translation task. The algorithm takes as input:

1. The feature-vector representation function $\phi(\cdot)$, that maps a pair $(\tau, a)$ to a real valued vector having any number of dimensions. Each dimension corresponds to a distinct feature function that maps: $S \times A_\tau \to \mathbb{R}$. Details of the features implemented for our model are given in section 3.8.

2. The DMT task formalized as $T \equiv \{S, \{A_s\}, T, R, \gamma\}$, this mapping has been described at the beginning of section 3.6.3.

3. The data $D$ consists of a set of pairs of sentences, $D \equiv \{(x, t^*)\}$, where $x$ is the source sentence and $t^*$ is the reference translation.
The trainer returns the weight vector, $\mathbf{w}$.

The main loop between lines 2-11, runs as many training epochs as needed to reach convergence. The inner loop, lines 3-10, runs a single training epoch over all the data set $D$, by taking in consideration a single sample $(x, t^*) \in D$ at each iteration. At line 4, the state $\tau$ is sampled from a uniform distribution over $\text{path}(s_0, \pi_\mathbf{w}) = \langle s_0, s_1, \cdots, s_\sigma | \pi_\mathbf{w} \rangle$, where $\pi_\mathbf{w}$ is the policy determined by the current weight vector $\mathbf{w}$. At line 5, $\hat{a}$ is set to be the action that would be preferred by the current policy at the state $\tau$. At line 6, the action $a'$ is sampled from a Zipfian distribution over the elements of $\{A_\tau - \hat{a}\}$ sorted with the $\tilde{Q}_{\pi_\mathbf{w}}(s, a)$ function, meaning that $a'$ is drawn from the set of possible actions $A_\tau$ excluding the current top scoring action $\hat{a} = \pi_\mathbf{w}(\tau)$. In this way, actions with a higher score have a higher probability to be drawn, while actions at the bottom of the rank still have a small probability to be selected. $a'$ represents a deviation from the path determined by the current policy.

The if statement at line 7 tests if the translation produced by $\text{path}(T(s, a'), \pi_\mathbf{w})$ has higher BLEU score than the one produced by $\text{path}(T(s, \hat{a}), \pi_\mathbf{w})$. The notation $\text{BLEU}(t^*, t')$ stands for the BLEU score produced by comparing the gold standard translation $t^*$ with the output translation $t'$. The notation $t(s, a)$ stands for the output translation produced by the current policy $\pi_\mathbf{w}$ starting from the state $T(s, a)$. This test corresponds to the if at line 6 of DRL meta-algorithm described in Algorithm 10, that is testing if $Q^*_{\pi_\mathbf{w}}(s, \hat{a}) < Q^*_{\pi_\mathbf{w}}(s, a')$. This can be derived considering the definition of $Q^*_{\pi}(s, a)$:

$$Q^*_{\pi}(s, a) = R(s, a) + \gamma V^*_{\pi}(T(s, a))$$

$$= R(s, a) + \gamma \sum_{s_i \in \text{path}(T(s, a), \pi)} \gamma^i R(s_i, \pi(s_i))$$

and considering that for the DMT model we set $\gamma = 1$ and the reward function $R(\cdot)$ has been chosen to be a pure-delayed reward function implemented as the BLEU score with respect the reference translation $t^*$. We have that:

$$Q^*_{\pi}(s, a) = \text{BLEU}(t^*, t(s, a))$$

For the update statement at line 8 we use the Averaged Perceptron technique of Freund and Schapire (1999). Algorithm 11 can be easily adapted by modifying
3.7 Comparison between GSP and DRL

So far we have presented the DMT framework, as a novel approach to Machine Translation, and more generally to Structured Prediction. We proposed two distinct frameworks to train a DMT model: Guided Structured Prediction (GSP) and Discriminative Reinforcement Learning (DRL). In this section we discuss some of the core differences and similarities of these two training frameworks.

Both the proposed training frameworks use a Perceptron-like linear classifier to select the actions applied to incrementally build the translation. GSP aims to produce a translation that exactly matches the gold standard, while DRL aims to produce the translation that has the highest BLEU score compared to the gold standard. GSP triggers an update as soon as the partial structure does not match the gold standard. Thus updates are performed at an intermediate step of the decoding, when a full structure has not yet been generated. Instead, DRL requires a comparison among complete translations to compute the BLEU score. GSP updates promote/penalize a single action, while DRL updates promote/penalize sequences of actions that produced a consistent translation tree. GSP directly compares the produced output to the gold standard, while DRL compares two possible outputs from the search space and tests if the internal scoring function produces an ordering that is consistent with the loss function (BLEU score).

GSP has the constraint of having to match exactly the gold standard translation. But, given a constrained translation grammar, for many examples in the training set it is not possible to produce an exact match of the gold standard translation. This may happen because a specific lexical translation or word reordering cannot be produced using the rules included in the specific translation grammar that is being used by the model. Thus, for some training examples, GSP can only test the selection of a first portion of the sequence of actions. Thus, the GSP trainer can find itself in a state where no actions can extend the current
Chapter 3. Deterministic Hierarchical Machine Translation

partial structure to match the gold standard. At such a state, the training must be interrupted, and the translation of the rest of the sentence is skipped. This constraint means that GSP is not able to fully exploit the training set. This constraint does not look optimal for a task like MT, where there are often more than one correct translation for the same sentence. In contrast, DRL triggers updates using a loss function that quantifies how distant the proposed translation is from the gold standard.

3.7.1 Update techniques Comparison

Both the proposed training frameworks use a discriminative function to score and select the sequence of actions applied to incrementally build the translation. In preliminary experiments, we found that both the models gain from the application of the Averaged Perceptron technique (Freund and Schapire 1999). We also found that any other more aggressive update technique, such as Passive-Aggressive (Crammer et al. 2006), Aggressive (Shen et al. 2007), or MIRA (Crammer and Singer 2003), leads to worse accuracy. Simpler updates are largely more effective for both our models. For GSP, if an incorrect action has the top score, the training algorithm just does a simple update and moves on to the next training sample. While DRL compares the sampled structures and performs a simple update without forcing the structure with lower loss to have a higher score after the update. In preliminary experiments, using simple updates instead of iterative aggressive updates as described in (Shen et al. 2007) leads to a $\sim 10\%$ relative gain in BLEU score accuracy. Also, using simple updates instead of Passive-Aggressive updates Crammer et al. (2006) leads to a $\sim 20\%$ relative gain in BLEU score.

To understand this finding, consider that any MT decoder needs to learn to construct structures $(t, h)$, while the training data specifies the gold translation, $t^*$, but it gives no information on the hidden-correspondence structure, $h$. As discussed in Liang et al. (2006), there are output structures that match the reference translation using a wrong internal structure (e.g. assuming wrong internal alignment). While in other cases the output translation can be a valid alternative translation but gets a low $BLEU$ score because it differs from, $t^*$, aggressively promoting/penalizing structures whose correctness can be partially verified can be expected to harm generalization ability and thus lowers the accuracy.

This result is in contrast with the results obtained by discriminative training
techniques designed for other deterministic structured prediction models such as Guided Learning (Shen et al. 2007) and the discriminative training applied for Non-Directional Dependency Parsing (Goldberg and Elhadad 2010). For these models aggressive update techniques are effective while simple updates are largely more effective for our model. This is due to the fact that those models have been applied to fully-supervised tasks such as POS Tagging or Syntactic Parsing. Thus, we can expect that if the frameworks we propose are applied to fully-supervised tasks, aggressive updates techniques may be effective.

3.8 Undirected Features

In this section we show how the features designed for standard bottom-up HMT can be adapted to the undirected approach, and we introduce a new feature from the new class of undirected features that are made possible by the undirected approach.

3.8.1 Local Features

Local features (also known as context-free features) depend only on the action rule \( r \), and do not need context information to be computed. These features can be used in the undirected approach without adaptation, since they are independent of the surrounding structure. For our experiments we use a standard set of local features:

- The log-probability of the source phrase given the target phrase
- The lexical translation log-probabilities of the source words given the target words
- The lexical translation log-probabilities of the target words given the source words
- The Word Penalty feature

3.8.2 Contextual Features

Contextual features are dependent on the interaction between the action rule \( r \) and the available context provided by the partial structure \( \tau \). The context avail-
able in a standard bottom-up HMT decoder is the full information about the children’s yield, but no information about the head structure is given. While the context available in an undirected approach can be any partial or complete information about bottom-up or top-down structure. In DMT all the needed information about the available context is stored in the carry $\kappa_i$ which is associated with the postcondition $p_i$, specified by the action $a$. Therefore, the computation of contextual features whose carry’s size is bounded (like the LM) requires constant time, even if it is conditioned on the full $\tau$ structure.

**Language Model**

The adaptation of the LM contextual feature to the undirected case computes the scores of the new $n$-grams formed by adding the terminals of the action rule $r$ to the current partial translation $\tau$. In the case that the action rule $r$ is connected to $\tau$ via a child non-terminal, the carry is expressed as $\kappa_i \equiv ([W_L \ast W_R])$. Where $W_L$ and $W_R$ are respectively the left and right boundary target words of the span covered by $\tau$. This notation is analogous to the standard star notation used for the bottom-up HMT decoder (e.g. Chiang (2007) Section 5.3.2). In the case that $r$ is connected to $\tau$ via the head non-terminal, the carry is expressed as $\kappa_i \equiv (W_R] - [W_L)$. Where $W_L$ and $W_R$ are respectively the left and right boundary target words of the surrounding context provided by $\tau$. The boundary words stored in the carry and the terminals of the action rule are all the information needed to compute and score the new $n$-grams generated by the connection action. A detailed example including a description of the LM features is described in section 3.4.2.

In the HMT bottom-up decoding framework, each partial structure covers a continuous span on both source side and target side. Thus, the bottom-up carry for an $n$-gram LM needs to store at most the $|2 \cdot (n - 1)|$ boundary words of the two span boundaries. A DMT partial structure $\tau$ does not cover a continuous sequence of words on the target side, and the total number of boundary words required to be stored is not bounded. However, each individual DMT non-bottom-up carry, $\kappa_i \in K$, still needs to store at most $|2 \cdot (n - 1)|$ boundary words for the LM. And each connection action considers only a bounded number of carries. Thus each connection action needs to deal with a bounded number of words. This implies that the computation of the LM factors for DMT models has the same complexity
3.8. Undirected Features

as standard HMT.

Context-Free Factor

In addition, we introduce the Context-Free Factor (CFF) features. CFF features are a kind of contextual feature, that can be used only in DMT models and not in standard HMT. CFF is an example of the set of features that can be introduced for undirectional models and cannot be used by a standard bottom-up model.

An action rule $r$ is connected to $\tau$ via one of $r$’s non-terminals, $X_{r,\tau}$. Thus, the score of the interaction between $r$ and the context structure attached to $X_{r,\tau}$ can be computed exactly, while the score of the structures attached to other $r$ non-terminals (i.e. those in postconditions) cannot be computed since these branches are missing. Each of these postcondition nonterminals has an associated CFF feature, which is an upper bound on the score of its missing branch. More precisely, it is an upper bound on the context-free component of this score. This upper bound can be exactly and efficiently computed using the Forest Rescoring Framework Huang and Chiang (2007); Huang (2008). This framework separates the MT decoding in two steps. In the first step only the context-free factors are considered. The output of the first step is a hypergraph called the context-free-forest, which compactly represents an exponential number of synchronous-trees. The second step introduces contextual features by applying a process of state-splitting to the context-free-forest, rescoring with non-context-free factors, and efficiently pruning the search space.

To efficiently compute CFF features we run the Inside-Outside algorithm with the $(\max, +)$ semiring (Goodman 1999) over the context-free-forest. The result is a map that gives the max Inside and Outside score for each node in the context-free forest. This map is used to get the value of the upper bound of the context-free factor of the score of the missing branches. This value is used as the CFF feature. CFF features can be computed in constant time while running the forest rescoring step by using the map computed on the context-free forest in a preprocessing step. CFF features account for the context free score of missing branches, helping estimating the future cost of partial derivations.

For bottom-up models, where the forest rescoring step is executed with Cube Pruning, it makes no sense to use CFF features, since the partial tree structures are compared only if they have the same missing head connection, so the CFF
feature would only add the same constant factor to all the elements to be compared. The CFF features make it easier to compare two alternative actions which would result in different missing components of the structure.

3.9 Experiments

So far, we have presented DMT, a novel deterministic model for machine translation. DMT allows translation trees to be produced in an undirectional manner. Furthermore we introduced two discriminative training frameworks that can be applied to learn the parameters of the DMT’s local discriminative decision function. In this section, we will compare these models with the standard HMT bottom-up model (Chiang 2007) trained with a standard Minimum Error Rate Training (MERT) (Och 2003). The systems are compared in terms of training and decoding runtime and BLEU score accuracy.

We implemented our model on top of Cdec (Dyer et al. (2010)). Cdec is known to be a fast, memory efficient and highly optimized C++ implementation of standard MT frameworks such as the forest rescoring framework. It provides a standard implementation of the HMT decoder and MERT training for translation hypergraphs. This allows us to directly compare the core of our deterministic approach with standard bottom-up Cube Pruning decoding in an equivalent context: using the same interface to the LM module, the same implementation of the rescoring-forest framework, the same I/O methods, etc.

Comparison of Features

The bottom-up HMT system uses exactly the same context-free features used by the DMT model (see Section 3.8.1). HMT and DMT use a different but analogous set of contextual features. HMT uses the standard bottom-up LM implementation, while DMT uses the LM adaptation for undirected models that we detailed in section 3.8.2 and we implemented within the Cdec toolkit. HMT does not use the CFF feature, which is only useful to undirected models. Despite these differences on the contextual features, the two models are still to be considered directly comparable, since LM features are used in both models in the version that is fitting the characteristics of each model, and the CFF features could be added to HMT but not produce any variation in the decoding process as explained in
3.9. Experiments

<table>
<thead>
<tr>
<th>Model</th>
<th>avg. sent. t.</th>
<th>total t. var.</th>
<th>rescoring t.</th>
<th>resc. t. var.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMT</td>
<td>135.2 ms</td>
<td>56.75%</td>
<td>38.9 ms</td>
<td>82.9%</td>
</tr>
<tr>
<td>HMT (\text{beam}=1)</td>
<td>232.0 ms</td>
<td>25.83%</td>
<td>141.3 ms</td>
<td>35.0%</td>
</tr>
<tr>
<td>HMT (\text{beam}=30)</td>
<td>312.8 ms</td>
<td>-</td>
<td>226.9 ms</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1: Decoding speed comparison.

section 3.8.2. Thus the set of contextual features used for HMT can be considered a bottom-up adaptation of the contextual feature set used for DMT.

Corpus and Experimental Setup

The experiments were executed on the NIST Chinese-English parallel corpus. This corpus is widely used and allows us to extend the comparison with all the MT models that have been tested on this corpus. The training corpus contains 239k sentence pairs with 6.9M Chinese words and 8.9M English words. The accuracy of the different systems is compared on the NIST 2003 test set, which contains 919 sentence pairs. The hierarchical translation grammar was extracted using the Joshua toolkit (Li et al. 2009) implementation of the suffix array rule extractor algorithm (Callison-Burch et al. 2005; Lopez 2007). The workstation used has Intel Core2 Duo CPU at 2.66 GHz with 4M of cache, 3.5 GB RAM, and it runs Linux kernel version 2.6.24 and gcc version 4.2.4.

3.9.1 Speed Comparison

We begin the model comparison by measuring the speed. The speed comparison is done in terms of algorithm run-time. We consider both training speed and decoding speed. The results show empirically that the proposed model greatly reduces the decoding and training time.

Decoding Speed

Table 3.1 reports the decoding speed measures. The DMT decoder is clearly the fastest among the compared models. The second column ("avg. sent. t." ) reports the average decoding time per sentence. We can notice that DMT is almost twice as fast than HMT with \(\text{beam} = 1\). The third column ("total. t. var"), reports the decoding time reduction relative to HMT with beam 30. Reducing the beam
<table>
<thead>
<tr>
<th>Model</th>
<th>avg. sent. t.</th>
<th>rel. t. red.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMT with GSP</td>
<td>143.1 ms</td>
<td>87.59%</td>
</tr>
<tr>
<td>DMT with DRL</td>
<td>267.4 ms</td>
<td>76.84%</td>
</tr>
<tr>
<td>HMT $\text{beam}=1$</td>
<td>765.2 ms</td>
<td>33.65%</td>
</tr>
<tr>
<td>HMT $\text{beam}=30$</td>
<td>1153.5 ms</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.2: Training speed comparison.

from 30 to 1 leads to 25.83% relative reduction in total decoding time. Since the HMT cannot run with beam lower than 1, this is the best performance in terms of speed for the HMT model. In contrast, DMT reaches a 56.75% relative time reduction in total decoding time, more than double the relative time reduction of HMT $\text{beam}=1$.

The fourth column (“rescoring t.”) reports the total time spent on the forest rescoring step. This is the step that involves the integration of the Language Model and other contextual features. The forest rescoring step is the part of the three decoders that actually differs. The remaining time is spent doing equivalent operations, like I/O for the same (or similar size) files, building and scoring the context-free forest, and converting data structures. We can notice that for standard HMT with beam 30, the forest rescoring step takes approximately two thirds of the total decoding time. Thus this is the most time consuming step in the pipeline. The fifth column (“resc. t. var.”), reports the relative variation in forest-rescoring time. Here we see that DMT reduces forest-rescoring time by almost a factor of 10 compared to HMT with beam 30. Running HMT $\text{beam}=1$ reduces the rescoring time by 35%, while DMT reaches 82.9% of relative time reduction compared to HMT $\text{beam}=30$.

Notice that for the decoding speed comparison there is no need to distinguish between DMT trained with GSP and DMT trained with DRL, since the decoding time is independent of the way the parameters of the model have been estimated.

**Training Speed**

The measures of the training time are reported in Table 3.2. These results show that the proposed model also greatly reduces the training time. In the second column (“avg. sent. t.”) we report the average training time per sentence for each model. These training times are in general higher than the decoding times shown.
3.9. Experiments

in Table 3.1. In the third column (“rel. t. red”) we report the relative training time reduction for each model. These relative advantages of DMT compared to HMT on training time is higher than the advantage when compared on decoding time. This is due to the higher level of complexity of the MERT technique employed to train HMT.

In general, the training algorithm of the discriminative models is often closely related to the decoding algorithm. A generic discriminative training model triggers updates of the weight vector based on the comparison between the training data and the output generated by the decoder. This is indeed the case for the two proposed training methods for DMT.

GSP executes the training on a sentence by running the decoding algorithm. As discussed in section 3.7, the GSP algorithm may not be able in some cases to complete the translation of the sentence because it can end up in a state where no correct actions are available to continue. Thus the average number of actions per sentence performed by the GSP trainer may be lower than the average number of actions per sentence performed at decoding. Despite that, the average training time per sentence is higher for the GSP trainer than for the decoder. This is due to the fact that GSP has to spend time in additional operations, such as testing if an action conflicts with the reference translation.

The DRL training algorithm triggers updates of the weight vector after comparing the output generated by the current policy with the output generated following a random deviation. Thus the most computationally expensive parts of the DRL algorithm are the two calls to the decoder. Notice that the two calls to the decoder differ only on the forest rescoring step. Thus all the other steps of the decoding pipeline can be executed only once. Additionally there are other minor operations to consider such as the two sampling steps and the update of the weight vector.

Both of the proposed training frameworks require approximately 10 training epochs to reach convergence.

3.9.2 Accuracy Comparison

The large execution time reduction comes at the cost of lower accuracy. The accuracy in terms of BLEU score is reported in Table 3.3. The second column (“BLEU”) reports the measured BLEU score for each of the compared system.
<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>relative loss</th>
<th>significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMT with GSP</td>
<td>29.73</td>
<td>7.61%</td>
<td>0.13</td>
</tr>
<tr>
<td>DMT with DRL</td>
<td>30.14</td>
<td>6.33%</td>
<td>0.18</td>
</tr>
<tr>
<td>HMT $\text{beam}=1$</td>
<td>30.87</td>
<td>4.07%</td>
<td>0.21</td>
</tr>
<tr>
<td>HMT $\text{beam}=30$</td>
<td>32.18</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.3: Accuracy comparison in terms of $\text{BLEU}$ score. “Significance” is the one-tail $p$-value.

DMT trained with DRL loses 2.04 $\text{BLEU}$ points compared to HMT with beam 30. While DMT trained with GSP loses 2.45 $\text{BLEU}$ points compared to HMT with beam 30. This variation corresponds to a relative-loss of 6.33% for DRL and 7.61% for GSP, as we report in column three (“relative loss”). This variation is not considered big (e.g. at the WMT-11 Machine Translation shared task Callison-Burch et al. (2011)).

To measure the significance of the variation, we compute the sign test and measure the one-tail $p$-value for the presented models in comparison to HMT $\text{beam}=30$. From the values reported in the fourth column we can observe that the significance of the BLUE score variations is not big, (e.g. at WMT-11 two systems were considered equivalent if $p>0.1$, as in these cases).

DMT trained with DRL reaches a higher accuracy than DMT trained with GSP. This shows that training the DMT model to maximize a global loss function can give better performance. In this cases the global loss function that we optimize is the $\text{BLEU}$ score itself, thus this result may not hold if we evaluate with a different metric.

The accuracy cannot be compared in terms of search score as we did for the Cube Pruning Algorithms in section 2.8, since the models we are comparing are trained with distinct algorithms and thus the search scores are not comparable.

To test the impact of the CFF features, we trained and tested the proposed training frameworks without these features. We measured that, for DMT with DRL the accuracy decrease by 1.8 $\text{BLEU}$ points, while for DMT with GSP the accuracy decrease by 2.3 $\text{BLEU}$ points. Thus these features are fundamental for the success of the greedy/deterministic approach. They provide an estimate of the score of the missing branches, thus helping to avoid some actions that have a good local score but lead to final translations with low global score. We believe that this feature is less important for DRL because this training framework is
3.9. Experiments

Figure 3.3: Speed/Accuracy balance reached by the CP algorithms at different values of the beam size.

designed to assign a score to each action that approximates its future cost, while GSP has no other mechanism to approximate the future cost other than the CFF feature.

3.9.3 Speed/Accuracy Balance

As we showed so far, DMT models are faster than standard HMT. But this speed gain comes at the cost of some accuracy loss. Thus DMT models offer a speed vs. accuracy balance that was not previously offered by HMT models. In Figure 3.3 we plot a scatter that represents the speed/accuracy balance achieved by the compared models. Considering that HMT beam = 1 is the fastest possible configuration that this model can reach, we can see that DMT models are located in a portion of the plane that cannot be covered by HMT models. This achievement is reached without deteriorating too much the accuracy of the system.
3.10 Discussion and Future Work

In general, we believe that deterministic solutions are promising for tasks like MT, where there is not a single correct solution: normally there are many correct ways to translate the same sentence, and for each correct translation there are many different derivation-trees generating that translation, and each correct derivation tree can be built greedily following different inference orders. Therefore, the set of correct decoding paths contains a number of elements that grows exponentially with the sentence size. This setting gives to a well-designed deterministic algorithm a chance to find a good translation even using a greedy policy.

As we discussed, the presented DMT decoder has more freedom in the choice of the actions that can be performed to build a translation. However, every chosen action can never be retracted. Thus, a DMT model cannot recover from an erroneous choice. HMT models solve this problem by allowing a beam search that is made efficient through the use of dynamic programming techniques. The DMT decoder could be extended allowing a beam search or allowing the possibility to retract a segment of the path at an intermediate stage. These techniques would increase the complexity of the decoder but increase its accuracy, resulting in a different speed/accuracy balance.

To take advantage of the full power of discriminative training and lower decoding complexity, it is possible to vastly increase the number of features. The DMT’s undirected nature allows better integration of contextual features that cannot be used by standard HMT and PbMT.
In addition to the related works discussed in the Introduction, and in the two main sections of the thesis, in this section we list in chronological order the publications that have been most influential in inspiring the development of the lines of work described in this dissertation.

**Literature review**

*(Magerman 1995)*

The model proposed in (Magerman 1995) is representative of how structured prediction was applied to syntactic parsing in the early 90s. It shows that statistical methods can significantly outperform models based on manually encoded grammars, that had been the state of the art for parsing until then. The proposed model builds the parse tree incrementally, through a sequence of disambiguation decisions that choose between POS tags, constituent structures and constituent labels. The decoder is formalized as a decision tree, but it can actually be interpreted as a classifier. It uses lexical forms, POS tags, and the history of the previous decisions as features. The structure is built in a bottom-up left to right fashion. The parameters of the model are estimated using counts from the training set, and then smoothed using Deleted Interpolation. The proposed model is compared to the state of the art IBM parser based on a manually encoded grammar (Black et al. 1993). The comparison is evaluated using the 0-crossing bracket measure, which represents the percentage of sentences for which none of
the constituents violates the boundaries of any gold standard constituents. On
the Lancaster Treebank (Black et al. 1993) the IBM parser achieves a 0-crossing-
brackets score of 69% while the presented model scored 76%. On the standard
Penn Treebank corpus the model reaches 86% unlabeled and 84.2% labelled F-
measures.

(Klein and Manning 2001)
Considering that parsing can be viewed as a process of logical deduction, and
exploiting the deep connection between logic and the hypergraph formalism, this
paper presents an influential view of parsing as directed hypergraph analysis
which covers both symbolic and probabilistic parsing. Furthermore it presents
a hypergraph based Chart Parser for arbitrary PCFGs that uses an Early-like
implicit binarization of rules and therefore offers the same cubic time bound as
a categorical chart parser and also allows a variety of grammar encodings and
induction strategies (bottom-up, top-down, left corner).

(Collins 2002)
This work successfully adapts the perceptron algorithm to structured prediction.
The task addressed is sequence labelling. The generation of label sequences is
modelled as a Hidden Markov Model (HMM). The model relies on Viterbi de-
coding, thereby performing an exhaustive search of the space of solutions. The
training algorithm is based on the perceptron simple additive updates, with av-
eraging technique. The paper includes a proof of convergence of the perceptron
algorithm adapted to structured prediction. Compared to Conditional Random
Fields (CRF), the perceptron training is more efficient since it does not need to
explicitly compute a partition function. The experimental results show improve-
ments over a Maximum Entropy (ME) tagger, 11.9% relative reduction in error
for POS tagging and a 5.1% relative reduction in error for NP-chunking.

(Koehn et al. 2003)
Koehn et al. (2003) propose a new phrase-based translation framework that en-
ables to the evaluation and comparison of several phrase-based translation mod-
els. A large number of experiments are carried out to understand better and
explain why phrase-based models outperform word-based models. The empirical
results suggest that the highest levels of performance can be obtained through relatively simple means: heuristic learning of phrase translations from word-based alignments and lexical weighting of phrase translations. Learning phrases from high-accuracy word-level alignment models does not have a strong impact on performance. Learning only syntactically motivated phrases degrades the performance of the phrase-based systems. Learning phrases longer than three words improves performance little for training corpora of up to 20 million words, suggesting that the data may be too sparse to learn longer phrases. Phrase-based models can robustly perform translations that are localized to phrases that are common enough to have been observed in training.

(Klein and Manning 2003)
This paper presents an extension of the classic A* search procedure to tabular PCFG parsing. The use of A* search dramatically reduces the time required to find a best parse by conservatively estimating the probabilities of parse completions. Estimates based on context and grammar summarization-features are discussed and efficient algorithms are proposed for computing them in preprocessing steps and integrating them in the heuristic search. The presented A* method is guaranteed to find the most likely parse, achieving the kind of speed-up achieved by best-first and finite-beam approximations. The presented A* parsing algorithm maintains worst-case cubic time.

(Koehn 2004)
This paper describes Pharaoh, a freely available decoder for Phrase-based Statistical Machine Translation models. The decoder is based on an efficient implementation of a dynamic-programming search algorithm with lattice generation.

(Collins and Roark 2004)
This paper describes an incremental parsing approach where parameters are estimated using a variant of the perceptron algorithm. The training and decoding problems are very closely related. The training method decodes training examples in sequence, and makes simple corrective updates to the parameters when errors are made. The decoder is based on an incremental, left-to-right parser, to find the highest scoring analysis under the model. A beam-search algorithm is used
during both training and decoding phases of the method. Experimental results show that the proposed model reaches competitive performances compared to a generative model that uses the same feature set on parsing the Penn Treebank.

Furthermore, they integrate the generative and discriminative models, obtaining a hybrid system. This is obtained by training the discriminative model using the negative log probability produced by the generative model as an additional feature. Experiments show that the hybrid model obtains an improvement of 2.1% over the generative model alone, reaching a total F-measure accuracy of 88.8%. This approach is an extremely simple method for integrating additional features and/or models into a discriminative model: essentially all that is needed is a definition of a feature-vector representations of entire parse trees, and then the existing parsing algorithms can be used for both training and decoding with the models.

(Daumé III and Marcu 2005)
This paper introduces a framework for learning as search optimization. It proposes to consider the learning problem and decoding problem in an integrated manner. The purpose is to avoid the need to heuristically combine an a priori unrelated learning technique and search algorithm, that is often the case for many complex problems for which search and parameter estimation cannot be performed exactly. Two online parameter update methods are detailed: a simple perceptron-style update and an approximate large margin update. The framework is tested on a simple syntactic chunking task for which exact search is possible and a joint tagging/chunking task for which exact search is intractable. The results show that the proposed model is able to outperform the state of the art. The best results are obtained with the large margin training method based on the ALMA algorithm (Gentile 2001)

(Chiang 2005)
This paper presents the first Tree-based MT model that reached state of the art results previously established with Phrase Based models. Formally it is a Synchronous Context Free Grammar (SCFG). This model leverages the intuition that sentences in all languages are composed by constituents that are recursively and hierarchically generated. This kind of model is formally syntax-based, but
in practice it makes no use of any actual syntactic information. Differently from
the Phrase-based models that only allow to locally reorder words in a window of
fixed size, this approach allows one to have a reordering model on a sentence level.
The grammar is extracted using information provided by an external aligner. The
rules are scored using heuristics. The rules extracted are binary. A single non-
terminal value is allowed in the grammar. To reduce the spurious ambiguity, the
grammar is filtered using heuristics. The decoding is executed with CKY parser
beam search. The attempt to introduce features that use syntactic information
gave no statistically significant improvement. The comparison with the Phrase-
based model Pharaoh on the Chinese-English MT task shows to an improvement
of 2 BLEU points.

(Zhang et al. 2006)
In this paper some of the earlier results published for Synchronous Context Free
Grammars and Tree Transducers models used for MT are summarized. The work
focuses on the need to apply a correct binarization of the extracted rules to reduce
the algorithm complexity, and on how this reduction of complexity supports a less
aggressive pruning strategy that results in improved accuracy of the results. The
parallelism between SCFG and Tree Transducer (TT) models is highlighted along
the whole presentation, TT is described as a more expressive version of SCFG,
allowing them to draw general conclusion for both kinds of models. The main
contribution of the paper is the formal description of a fast binarization algo-

(Liang et al. 2006)

This work presents a discriminative approach to Phrase-Based Machine Transla-
tion. Different error-driven approaches to train the perceptron-based decoder are
discussed and tested. Making frequent but smaller updates toward the transla-
tion in the best-k with highest BLEU-score is more efficient than using a standard
Perceptron update to match the target translation. This finding is motivated by
different reasons: 1) The Perceptron update cannot exploit sentences for which
the target translation is not reachable, 2) Often the standard perceptron update promotes translations matching the target translation but having a wrong internal structure (based on wrong alignment), penalizing a reasonable translation that is not matching the target translation, 3) Updating toward a translation with higher BLEU score partially biases a BLEU score based evaluation. In addition, a set of features is discussed and tested: 1) Features that learn weights for individual part-of-speech patterns; 2) Alignment constellation features that weight phrases given the word alignment pattern that led to their extraction; 3) And POS features to parametrize the distortion model.

(Chiang 2007)
This paper expands on (Chiang 2005) with a more detailed description of the model and of the Cube Pruning based decoding algorithm, and includes additional experimental results. The presented hierarchical model achieves a significant improvement in accuracy over a state of the art Phrase-based system for language pairs requiring complex word reordering such as English-Chinese.

(Shen et al. 2007)
This paper presents a novel learning framework for sequence classification named Guided Learning (GL). GL lifts any constraint on the inference order, e.g. left-to-right, and allows the model to learn and induce an inference order based on an easiest-first strategy. As a result the model starts building the label sequences from the easy decisions, leaving the difficult decisions for the end when more context is available. The absence of any constraint on the order of inference allows the use of a new kind of feature that conditions on the left and right context jointly. The task of learning the order of inference and training the local classifier are dynamically incorporated into a single Perceptron-like learning algorithm. The experimental results on POS tagging show that GL obtains an error rate of 2.67% on the standard PTB test set, which represents a 3.3% relative error reduction over the previous best result.

(Huang and Chiang 2007)
Huang and Chiang (2007) address the issue of efficient decoding in machine translation. The integration of the Language Model is recognized to be essential for the
translation quality and is responsible for the exponential complexity factor. This work introduces the forest rescoring framework, that is fundamental to isolate the LM integration step in the decoding pipeline. The Cube Pruning algorithm (Chiang 2005) is the k-best approach used to approximate efficiently the forest rescoring step. Cube Pruning is generalized for Phrase-based and tree-to-string MT systems. Furthermore a faster variation of Cube Pruning is devised, called Cube Growing, which uses a lazy version of k-best parsing reducing k to the minimum needed at each node to obtain the desired number of hypotheses at the root. The proposed method achieve significant speed improvements over the conventional beam-search method at the same levels of search error and translation accuracy.

(Li et al. 2009)
This paper introduces a toolkit for statistical syntax based Machine Translation named Joshua. The release of this open-source software aims to remove the barrier of entry for other researchers, and makes it easy to duplicate and compare experiments. This toolkit provides open-source implementations of the algorithms required for synchronous context free grammar (SCFG) based MT models such as: chart parsing, n-gram language model integration, k-best extraction, beam- and cube-pruning as described in (Chiang 2007). The toolkit also includes an implementation of suffix-array grammar extraction (Lopez 2008) and minimum error rate training (Och 2003). Furthermore it provides a novel technique used to sub-sample the training corpus that aims to reduce the size of the corpus minimally affecting the information provided. Parallel and distributed computing techniques are exploited to make it scalable.

(Cohn et al. 2009)
Cohn et al. (2009) present a way to estimate weighted Tree Substitution Grammars for Syntactic Parsing with Bayesian non-parametric technique. On small data samples this model produces similar results to a state-of-the art parser, with the induction of smaller and simpler grammars. This is achieved using a prior distribution over the tree fragments that encourages sparsity and reuse of simple rules in a Bayesian non-parametric formulation. Gibbs sampling is used for the training phase, and the parsing of novel sentences is approximated us-
ing Monte Carlo techniques. From the experiments emerges the inadequacy of
PCFG to solve parsing, due to the lack of context information carried by TSG.
As in Blunsom et al. (2009), the authors compare two priors, the first one bases
the non-terminal expansions on uniform distributions, the second instead bases
the generative process of tree fragments on a rudimental PCFG, also in this case
obtaining better results with the second case.

(Hopkins and Langmead 2009)
This paper bring some clarity to the situation of recent MT decoding techniques,
providing a deeper understanding of these methods with the aim to provide theo-
retical support to ease combinations and adaptations to new use cases. Specifi-
cally it takes a closer look at Cube Pruning, showing that it is essentially equiva-
 lent to A* search on a specific search space with specific heuristics.

(Carreras and Collins 2009)
This paper describes a model for Syntax based MT from German to English
making use of syntactic annotation on the English side. The model leverages a
discriminative non-projective dependency parsing technique to have flexible re-
ordering, and adds a discriminative model that can condition on rich features of
the source language string. The model makes use of phrasal entries augmented
with subtrees that provide syntactic information, the phrases are combined using
generalized parsing TAG operation. External alignment and a phrasal lexicon
need to be provided respectively from a standard alignment model and a Phras-
based MT model. The problem of spurious derivations is solved by constraining
the generation tree on the source side without limiting the power of the distortion
model on the target side, resulting in an accuracy improvement. The decoding al-
gorithm is based on an adaptation of the classic Dynamic Programming technique
usually applied in Phrase-based models, and a beam search strategy is applied
to treat the exponential complexity of the decoding algorithm. The definition of
the function that selects the best-k hypotheses is critical for the success of the
method. The model shows improvements over Pharaon Phrase-based MT model
(Koehn 2004).

(Cohn and Blunsom 2009)
Cohn and Blunsom (2009) propose a Bayesian model of tree-to-string translation which induces smaller and more accurate grammars compared to the grammars extracted with a two-stage approach which employs separate word alignment and heuristics that are extension of those developed for Phrase-based models. The selection of an adequate prior over grammar rules allows the grammar to be described with as few rules as possible. This avoids the degeneration of EM models that place disproportionate probability mass on the largest rules. The grammar is generated and weighted in a single step, without resorting to an external alignment. Iterating on the parallel corpus with syntactic trees on the target side with Gibbs sampling, the model infers a distribution over the derivations of training instances, where the set of derivations collectively specify the grammar. So syntax is not just a means to have better distortion model and lexical selection but is also used in the induction of the bilingual grammar. In the Chinese-English translation task the model is compared with a heuristic grammar extraction technique. The results shows that the two systems extract a grammar with similar size, but the proposed Bayesian model generates shallower rules with fewer nodes, and achieves also a BLUE score improvement.

(Blunsom et al. 2009)

This paper presents a novel non-parametric Bayesian approach for inducing synchronous context free grammars for MT. The originality of this work lies in the distinction of different rule types. The design allows rules that generate or reorder non-terminals, and rules that emit terminals, these terminals have no upper bound in the maximum number of words but are proper phrases as intended in a classical Phrase-based MT system. Sparse priors are applied over the model parameters to encode the intuition that source phrases have few translations, and also addressing the problem of over-fitting when using long multi-words translations pairs. It is interesting the way that a rudimentary MT system such as IBM Model 1 or a simple Phrase Based Model are integrated as emission priors, resulting in relevant improvements of the performances over the uniform prior. Another contribution of this work is the induction of categories for the synchronous grammar using hierarchical Dirichlet process prior. The intuition behind the use of this technique is that automatically induced non-terminal symbols give synchronous grammar models increased power over single non-terminal system, but the final
results shows a slight improvement between the single class system and the 5 class version. For the evaluation of the parameters, they used a mean-field approximation that offers better scaling and convergence properties than a Gibbs sampler, at the expense of increased approximation. Throughout the whole paper the authors show confidence in the superiority of the Bayesian techniques over the Expectation Maximization models, supported by the comparison experiments executed on a synthetic synchronous grammar and on a Chinese-English translation task.

(Dyer et al. 2010)
This paper presents cdec, an open source general framework for SMT. It supports different MT approaches: word-based, phrase-based and tree-based. It can be used for decoding, aligning and training. The architecture relies on general frameworks such as the hypergraph formalism, forest rescoring framework and the semiring formalism. The decoding is based on a pipeline logic. It separates model-specific translation logic from general rescoring, pruning, and inference algorithms. Two training pipelines are provided: Viterbi Envelope Semiring Training (VEST), and Minimum Error Rate Training (MERT). Cdec is intended to provide a standard implementation to ease the deployment, test and expansion of state of the art SMT models.

(Goldberg and Elhadad 2010)
Presents a novel deterministic easiest-first non-directional dependency parsing algorithm. Dependency trees are built by recursively creating the easiest arcs in a non-directional manner. By “non-directional” they mean the removal of the constraint of scanning the sentence from left to right, typical of shift-reduce models, but this algorithm still builds the tree structures in a bottom-up fashion. Standard left to right parsers can only base decisions on the left context. In contrast, the non-directional algorithm allows the incorporation of features from already built structures both to the left and to the right. The parser learns both the attachment preferences and the order in which they should be performed. The linear model used to compute the actions score is trained with the Averaged Perceptron technique. Perceptron updates are triggered when an invalid action is chosen. In this case, the weight vector is updated by decreasing the weights of
the features associated with the invalid action and increasing the weights for the currently highest scoring valid action. The result is a deterministic, best-first, $O(n \log n)$ parser, which is significantly more accurate than best-first transition based parsers, and nears the performance of globally optimized parsing models.
This dissertation investigates and extends the state of the art for Tree-based Machine Translation. We successfully propose new algorithms and frameworks that offer an accuracy/complexity balance that was previously unavailable for HMT.

The first part of the thesis successfully extends the work on Cube Pruning. This algorithm is fundamental for many systems combining inexact beam search with dynamic programming such as Hierarchical Machine Translation.

We propose the Faster Cube Pruning algorithms. FCPs improve over the standard CP algorithm in terms of speed and memory usage, leveraging a more aggressive pruning of the search space and optimizing the use and maintenance of data structures. For each of the algorithms we provide a proof that they return the exact solution in a monotonic search space setting. The experiments on the HMT task show that FCPs algorithms speed up the decoding algorithm without reducing the accuracy.

We present the Linear Cube Pruning algorithm, which approximates the CP algorithm while running in linear time. We provide a proof that LCP returns the exact solution for a relaxed version of the original CP problem. The experiments applying the LCP algorithm to HMT show an actual asymptotic time advantage balanced by a bounded loss in accuracy.

We provide a formalization of the Cube Pruning problem that is as generic as possible to ease the recognition of problems that may benefit from the use of this set of algorithms. This generic formalization allows us to draw a connection
Chapter 5. Conclusion and Final Remarks

between the Cube Pruning problem and a classic algorithmic problem such as K-way merge. The formalization of the connection between these two problems allows ideas and possible solutions to be ported from one domain to another. As an extension to this line of work, it would be interesting to find applications of the K-way merge problem that can benefit from the use of the proposed CP algorithms. For example, if we consider that CP is a special case of the K-way Merge problem, the discovery of the LCP algorithm implies that there exists a linear time solution for a subset of the K-way Merge problems. And vice versa, the existing literature on K-way merge can provide us with ideas to be ported in the CP domain.

A similar reasoning can be done for any other problem that can be mapped to the CP problem. The CP pruning problem has already been linked to many existing problems such as alignment and $A^*$ search (see section 2.2), resulting in finding new solutions to existing problems (as in the case of alignment) or extending the CP framework (as in the case of $A^*$ search). As an example, one could think to test the presented algorithm for any application of the CKY algorithm, such as Syntactic Parsing.

Furthermore, we define a general technique that allows any CP algorithm to be extended to a search space having any number of dimensions, without increasing its original complexity. This technique can be ported to any CP algorithm. For this dissertation, this technique is fundamental to remove the assumption on the number of dimensions that we impose and to simplify the explanation and formalization of new algorithms (e.g. we define LCP assuming a bidimensional search space).

The second part of the thesis presents the first Deterministic framework for MT. This framework represents a totally new approach to MT. This model combines advantages given by the use of hierarchical synchronous-grammars with a more efficient decoding algorithm. It applies, for the first time, undirectional search jointly with deterministic/greedy decoding. DMT’s undirected nature allows us to design novel undirected features that better approximate contextual features, and to introduce a new class of undirected features that cannot be used by standard bottom-up decoders. The experiments on the HMT task show that this decoding approach is significantly faster than CKY-based bottom-up decoding.
The DMT decoder could be extended by allowing a beam search or by allowing the possibility to retract a segment of the path at an intermediate stage. These techniques would increase the complexity of the decoder but increase its accuracy, resulting in a different speed/accuracy balance.

We define a discriminative training framework to learn the parameters of the novel Deterministic MT decoder. We prove that training the model in a discriminative fashion to match the sequence of correct actions individually without optimizing a global loss function can produce results of good quality even if a greedy search approach is used.

Furthermore, we apply Reinforcement Learning techniques to define an additional training framework for the DMT decoder. This RL based training approach focusses more on optimizing a global loss function using Reinforcement Learning techniques. The DRL framework is defined in general terms, such that it can be applied to train a discriminative scoring function for any structured prediction task.

To take advantage of the full power of discriminative training and lower decoding complexity, it is possible to vastly increase the number of features, in particular adding more of those undirected features, such as the Context-Free Factors, that can be applied only to undirectional models such as DMT.

In general, we believe that deterministic solutions are promising for tasks like MT, where there are usually many correct decoding alternatives. Normally there is not only a single correct translation, but many correct ways to translate the same sentence. Plus, for each correct translation there can be many different derivation-trees generating that translation. And each correct derivation tree can be built greedily following many different inference orders. For these reasons, the set of correct decoding paths contains a number of elements that grows exponentially with the sentence size. This setting gives a well-designed deterministic algorithm a chance to find a good translation even using a greedy policy.


of the North American Chapter of the Association for Computational Linguistics, New York, New York.